Hayes
Michie
and
Richards

Machine Intelligence 11

Oxford

# MACHINE INTELLIGENCE 11

# MACHINE INTELLIGENCE 11

Logic and the acquisition of knowledge

edited by

J. E. HAYES

*Research Associate,
Turing Institute*

D. MICHIE

*Chief Scientist,
Turing Institute*

and

J. RICHARDS

*Head of Industrial Studies,
Turing Institute*

CLARENDON PRESS · OXFORD
1988

*Oxford University Press, Walton Street, Oxford OX2 6DP*

*Oxford New York Toronto*
*Delhi Bombay Calcutta Madras Karachi*
*Petaling Jaya Singapore Hong Kong Tokyo*
*Nairobi Dar es Salaam Cape Town*
*Melbourne Auckland*

*and associated companies in*
*Berlin Ibadan*

*Published in the United States*
*by Oxford University Press, New York*

*British Library Cataloguing in Publication Data*

*Machine Intelligence.*
*1. Artificial Intelligence*
*I. Title II. Michie, Donald III. Richard J.*
*006.3*
*ISBN 0-19-853718-2*

*Library of Congress Cataloging in Publication Data*
*Data available*

*Typeset and printed in Northern Ireland at The Universities Press (Belfast) Ltd.*

# PREFACE

Held at intervals in Scotland, the first seven International Machine Intelligence Workshops spanning the period of 1965–71 were involved in developing the new subject internationally—in those early days mainly as a mid-Atlantic phenomenon. Japan and continental Europe had yet to enter in strength. Also in the wings was the ill-famed 'Lighthill report' which in 1973 stigmatized machine intelligence as a mirage and in the UK demolished its local infrastructure.

Two and a half millennia ago, the historian Thucydides observed that it is not fortifications which make a city but people. In spite of dispersion, the AI culture under challenge evinced both hardiness and solidarity. Included in the exodus from Britain's 'AI winter' were the MI Workshops themselves. Successively they found hospitality in Santa Cruz, USA (1975), Repino, USSR (1977), and Cleveland, USA (1981), by which time the distant tidings of Japan's Fifth Generation presaged the coming thaw. Preparations were begun to found a new UK centre, the Turing Institute at Glasgow. By 1985 sufficient critical mass existed for the new Institute to be able to host a return after fourteen years to the series' land of origin. With additional support from the University of Strathclyde, the eleventh Workshop took place at the University's study centre at Ross Priory near the banks of Loch Lomond.

The titles of the twenty papers which now emerge are indicative of a continuing trend towards unity of approach. Logical models of deductive and inductive reasoning become ever more central and find a common frame in interactive environments for practical problem solving. We also see the first demonstrations that the fruits of past solutions can be systematically digested by an automated solver and built into incremental bodies of new, human-type, knowledge.

The long expected maturation of machine intelligence is evidently at last occurring apace. An adolescent's elders not uncommonly warn, as elders of the physical sciences have of AI, that the youth may have outgrown his own strength. Has the maturation of machine intelligence been of this kind? With some confidence we commit this eleventh volume to the hands of its readers and invite them to pursue the question to their own conclusions.

*February 1988*

Donald Michie
Editor in Chief

# CONTENTS

# COMPUTATION AND LOGIC

1

# Partial Models and Non-monotonic Inference

K. Konolige
Artificial Intelligence Center, SRI International and
Center for the Study of Language and Information, Stanford University, USA

**Abstract**
The non-monotonic character of common-sense reasoning is well recognized, as we often jump to conclusions that are not strictly justified by our partial knowledge of a situation. Most formalizations of this idea are best described as syntactic transformations on theories, with little or no semantic underpinnings. In this paper we develop a method of non-monotonic reasoning from a strictly semantic viewpoint, namely, as conjectures about how the missing information in a partial model should be filled in. The advantages of this approach are a natural and intuitively satisfying formalization of diverse types of non-monotonic reasoning, among them domain closure, the unique names hypothesis, and default reasoning.

## 1. INTRODUCTION

The importance of non-monotonic reasoning for common-sense domains is widely recognized in the field of Artificial Intelligence (AI). In this paper we will be concerned with such reasoning in its most general form, that is, in inferences that are *defeasible*: given more information, we may retract them.

The purpose of this paper is to introduce a form of non-monotonic inference based on the notion of a *partial model* of the world. We take partial models to reflect our partial knowledge of the true state of affairs. We then define non-monotonic inference as the process of filling in unknown parts of the model with *conjectures*: statements that could turn out to be false, given more complete knowledge. To take a standard example from default reasoning: since most birds can fly, if Tweety is a bird it is reasonable to assume that she can fly, at least in the absence of any information to the contrary. We thus have some justification for filling in our partial picture of the world with this conjecture. If our knowledge includes the fact that Tweety is an ostrich, then no such justification exists, and the conjecture must be retracted.

Of course, there are many different ways to represent partial knowledge of the world; in AI, first-order theories (FOTs) are a widely used method. However, FOTs are in a sense *too* partial for the purpose of non-monotonic inference—it is often difficult to decide just how the 'partial' should be filled. For example, consider the sentence

$$Bird(Tweety) \vee Bird(Opus) \qquad (1.1)$$

This sentence gives us partial information about the world, in the sense we know either Tweety or Opus (or both) is a bird; but given just (1.1) it is impossible to conclude that we know Tweety to be a bird, or that we know Opus to be a bird.

Now suppose we are given a default rule stated informally as

*In the absence of conflicting information, assume that a bird flies.* (1.2)

How can this rule be applied to our bird theory (1.1) to make conjectures about the ability of Tweety and Opus to fly? One approach is to relate the application of the default to a consistency condition on the theory, as in the default theories of Reiter (1980). Roughly speaking, our informal rule translates into the following rule for extending an FOT:

*If in a theory x is a bird and it is consistent to assume that x can fly, do so.* (1.3)

Unfortunately such a default rule yields no new information when applied to (1.1). The disjunction does not permit us to conclude that any particular individual is a bird, and so it is impossible to instantiate the variable $x$ in the antecedent of the default rule.

But clearly our intuitions are that (1.2) tells us something more about the theory (1.1). Suppose we ask what possible partial states of affairs would make (1.1) true. One of the following two is a minimally necessary condition:

1. Tweety is a bird.
2. Opus is a bird.

Now the application of the default rule is straightforward for each case, so we conjecture that either Tweety or Opus can fly.

One conclusion to be drawn from this example is that default reasoning should be based on an analysis of the models that a theory admits. It is the claim of this paper that partial models are an appropriate and natural level of description for the application of default rules, and other types of non-monotonic reasoning as well. In the next section, we support this claim by discussing general principles for implementing non-monotonic reasoning as conjectures on partial models, and by criticizing another model-based framework for non-monotonic inference, McCarthy's (1980, 1984) circumscription schema, from this point of view. The rest of the paper is devoted to illustrating the general principles using a particular type of partial model based on Hintikka interpretations, defined in Section 3. Because these models use the constants of a theory as their domain, they admit very natural treatment of assumptions involving equality and the naming of individuals, which are illustrated in Section 4, along with other types of default reasoning, including domain closure and the assumption of disjoint domains.

## 2. A SEMANTICS FOR NON-MONOTONIC INFERENCE

In this section we consider some general principles of a partial-model approach to non-monotonic inference, and introduce notation to be used throughout the paper. An analysis of circumscription based on these principles is also presented.

### 2.1. Conjectures on partial models

Any consistent set of sentences (or *theory*) $T$ in a first-order language is satisfied by a set of (first-order) models. To continue the example from the Introduction: let *Tweety* refer to the individual TWEETY, and *Opus* to OPUS, and let BIRD and FLY be the properties of being a bird and flying, respectively. Now consider the models of *Bird*(*Tweety*) ∨ *Bird*(*Opus*):

$$
\begin{array}{lll}
M_1 = \text{BIRD}:\{\text{TWEETY}\} & \text{FLY}:\{\} & \\
M_2 = \text{BIRD}:\{\text{TWEETY}\} & \text{FLY}:\{\text{TWEETY}\} & \\
M_3 = \text{BIRD}:\{\text{TWEETY}, e_1\} & \text{FLY}:\{\} & \\
M_4 = \text{BIRD}:\{\text{TWEETY}, e_1\} & \text{FLY}:\{\text{TWEETY}\} & \\
M_5 = \text{BIRD}:\{\text{TWEETY}, e_1\} & \text{FLY}:\{\text{TWEETY}, e_1\} & \\
M_6 = \text{BIRD}:\{\text{TWEETY}, e_1, e_2\} & \text{FLY}:\{\} & \\
\quad\vdots & & \\
M_i = \text{BIRD}:\{\text{OPUS}\} & \text{FLY}:\{\} & (2.1) \\
M_{i+1} = \text{BIRD}:\{\text{OPUS}\} & \text{FLY}:\{\text{OPUS}\} & \\
M_{i+2} = \text{BIRD}:\{\text{OPUS}, e_1\} & \text{FLY}:\{\} & \\
M_{i+3} = \text{BIRD}:\{\text{OPUS}, e_1\} & \text{FLY}:\{\text{OPUS}\} & \\
M_{i+4} = \text{BIRD}:\{\text{OPUS}, e_1\} & \text{FLY}:\{\text{OPUS}, e_1\} & \\
\quad\vdots & & \\
M_j = \text{BIRD}:\{\text{OPUS}, \text{TWEETY}\} & \text{FLY}:\{\} & \\
\quad\vdots & &
\end{array}
$$

These models naturally fall into two groups, corresponding to one of the two disjuncts in the theory: either Tweety is a bird, or Opus is (there are models such as $M_j$ in which both these are true; such models fall into both groups). We can represent these groups by using the notion of a *partial model*. A partial model contains only a part of the information necessary in a (complete) model; by *extending* the partial model, we arrive at a set of models. In this example, we could construct two partial models by specifying just a part of the extension of the Bɪʀᴅ relation:

$$\begin{aligned} m_1 &= \text{TWEETY} \in \text{BIRD} \\ m_2 &= \text{OPUS} \in \text{BIRD}. \end{aligned} \tag{2.2}$$

The extension of $m_1$ includes $M_1 - M_5$ and $M_j$; the extension of $m_2$ includes $M_i - M_{i+4}$ and $M_j$. We write $E(m)$ for the set of extensions of a partial model $m$.

We have in $m_1$ and $m_2$ a formal model-theoretic counterpart of the informal reasoning we carried out in the Introduction. We can formulate the default rule 1.2 as the following conjecture:

*If within a partial model x is a bird and it is consistent to assume that x flies, do so.* (2.3)

Note that this is exactly the default rule (1.3), except that 'theory' has been changed to 'partial model'. A proposition $P$ is *consistent* with a partial model $m$ if there is an extension of $m$ satisfying $P$. In the case of $m_1$ there are models in which Tweety flies, and so (2.3) picks out just that subset $\{M_2, M_5, \ldots\}$ of $E(m_1)$; similarly, for $m_2$ we get the subset $\{M_{i+1}, M_{i+4}, \ldots\}$. Since the world could be described by either $m_1$ or $m_2$, we take the union $\{M_2, M_5, M_{i+1}, M_{i+4}, \ldots\}$ of these models as the result of default reasoning. Obviously, $Fly(Tweety) \vee Fly(Opus)$ is satisfied by each of these models.

To sum up: let $T$ be a theory and $\alpha$ a conjecture on partial models. A conjecture picks out a non-empty subset of the extensions of a partial model. Non-monotonic inference can be viewed as the following process:

1. Let $\mathbf{M}$ be all models of $T$. Form a set of all partial models $\mathbf{m}$. Let $\mathbf{M}'$ be $\mathbf{M} - E(\mathbf{m})$, i.e. all models not in the extension of some member of $\mathbf{m}$.
2. Let $C(\alpha, \mathbf{m})$ be the set of extensions of $\mathbf{m}$ chosen by the conjecture.
3. We say that a set of sentences $T'$ is *inferred by $\alpha$ from $T$* if every member of $T'$ is satisfied by each of $\mathbf{M}' \cup C(\alpha, \mathbf{m})$. We write this as $T \vdash_\alpha T'$.

*Remarks.* The general nature of non-monotonic inference here is the pruning of the set of models of a theory. For any given language $L$, we may have in mind certain types of models, the intended interpretations of $L$. For example, in studying resolution, we restrict our attention to Herbrand interpretations, in which all terms denote themselves.

We consider some general technical points of this definition. First, the inference operator $T\vdash_\alpha$ can be non-monotonic in $T$, as is easily shown by example. Let $\alpha$ be the conjecture that picks out only those extensions of a partial model in which $P$ is false. We have:

$$\{Q\} \vdash_\alpha \neg P \tag{2.4}$$

but

$$\{Q, P\} \nvdash_{\alpha} \neg P. \tag{2.5}$$

A special case, which is monotonic in $T$, is the conjecture $\delta$ that picks out *all* extensions of a partial model. The operator $\vdash_{\delta}$ is simple logical deduction, that is, for $T \vdash_{\delta} T'$, $T'$ is the set of logical consequences of $T$, and hence also deductive consequences, by the completeness theorem for first-order logic.

Because conjectures pick out a subset of the possible models of $T$, the inference operator has the reflexive property

$$T \vdash_{\alpha} T. \tag{2.6}$$

Conjectures are thus appropriate for default reasoning or defeasible reasoning in general, where the initial facts, though sparse, are assumed to be accurate. There are, of course, other types of non-monotonic reasoning that are not naturally expressed as conjectures: for example, events are often treated formally as arbitrary transformations on models, and the revision of belief on the basis of new information requires changing a theory to admit models it did not originally have.

There is no guarantee that partial models exist, or if they do, that their extensions fully cover the set of models **M**. **M**′ is designed to take up the slack in these situations, so that all models of **M** are 'accounted for'. This, and the fact that conjectures are a pruning operation on sets of models, yield the following consistency property for the inference operator: if the initial theory $T$ is consistent, then any set of inferred sentences is also consistent; that is, it is impossible to have

$$T \vdash_{\alpha} p \wedge \neg p. \tag{2.7}$$

The notion of the *coverage* of partial models is an important one, and is in some sense a completeness criterion for this method. If there are *no* partial models for a given theory, then for every conjecture $\alpha$ the operator $\vdash_{\alpha}$ becomes logical deduction, and no non-monotonic inference takes place. If the partial models of a theory fully cover the intended models (that is, every intended model is an extension of some partial model), then a conjecture on the partial models takes into account all of the interpretations of the theory. For example, the two partial models (2.2) cover all the models of $T = Bird(Tweety) \vee Bird(Opus)$, and so the conjecture (2.3) gives us the maximum restriction on the models of $T$. An important feature of conjectures is that they degrade gracefully when some models are not covered, either because of theoretical or computational limitations. If for some reason only $m_1$ is used as a partial model of $T$, then the conjecture (2.3) produces the weaker result

$$T \vdash_{\alpha} Bird(Tweety) \supset Fly(Tweety). \tag{2.8}$$

One of the strengths of the method is that there are many different ways to construct partial models of the world. The type of partiality we choose to represent will influence the nature of the non-monotonic operator $\vdash_\alpha$. For example, we might take partial models to be a subset of each relation's (positive) extension, as we did in (2.2); data bases are often viewed in this way (Gallaire *et al.*, 1978). A partial model of this sort covers a set of models that agree on the common subset, but can otherwise disagree. It invites the conjecture that the subset is the complete extension: there are no other true positive facts about the world (sometimes referred to as the *closed-world assumption*; see Section 4).

An important type of partiality, and one we will exploit for most of the remainder of this paper, is the ability to leave unspecified the equality (or inequality) of terms in a theory. One way to do this is by introducing syntactic elements into the partial models, as we do with Hintikka sets in Section 3. Partial models then become sets of atoms and their negations, including equality predications. For example, the set $\{Bird(Tweety), Bird(Opus)\}$ has extensions in which Tweety and Opus are the same individual, and in which they are different. Assumptions about the uniqueness of named individuals can be framed in terms of conjectures on this partial model.

### 2.2. Circumscription

Predicate circumscription is a proof-theoretic technique in which an FOT $T$ is augmented by a circumscription formula. We can summarize its current formulation (from Etherington *et al.*, 1984) as follows: let $P$ be a predicate, and $P'$ a finite sequences of predicates of a finite theory $T$. Then $Circ(T, P, P')$ is a particular second-order formula expressing the circumscription of $P$, letting the predicates $P'$ vary.

The semantics of circumscription come from the notion of $P$-minimal models. A model $M$ is $P$-minimal if there is no other model $N$, agreeing with $M$ everywhere except for the predicates $P$ and $P'$, such that the extension of $P$ in $N$ is a proper subset of that in $M$. Circumscription is sound with respect to minimal models, in the sense that $Circ(T, P, P')$ is true in all $P$-minimal models of $T$; however it is known to be incomplete (these results are summarized in Minker and Perlis, 1985).

Partial model conjectures have close ties with reasoning about minimal models. In fact, we can express the intended semantics of circumscription as a conjecture in the following way. We take partial models to be the $P$-minimal models, where the extension of a partial model $M$ is the set of all models $N$ which agree with $M$, except possibly on $P$ and $P'$. The conjecture $\alpha$ is to pick only the minimal model itself.

Reasoning about minimal models was first employed in AI by McCarthy (1980) in an attempt to deal with what he called the *qualification problem*. In brief, this is the problem of stating formally what objects and conditions *do not obtain* in a given situation. Using minimal models is a means of applying Occam's razor: only those objects are assumed to exist that are actually required by the statements of a theory.

It is not clear, however, that reasoning in minimal models is the best means of performing defeasible reasoning in general. For example, it can lead to a complicated statement of defaults by means of an abnormality predicate. Compare the compact formulation of Example 4.9 with the corresponding circumscriptive rendering on pp. 300–302 of McCarthy (1984). But the evidence here is not yet in, and awaits a fuller exploration of the application of circumscription.

With regard to assumptions about equality, certain inherent limitations are already known (see Etherington *et al.*, 1984). Because minimal models are defined with respect to a fixed denotation function for the terms of a theory, it is impossible to perform non-monotonic inferences about the equality of terms by reasoning in such models. However, there have been attempts to account for equality by importing names and their denotations as objects of the domain (Lifschitz, 1984; McCarthy, 1984).

By contrast, we can choose partial models in such a way that non-monotonic inferences about equality are possible. As we show in the next section, partial model conjectures enjoy a natural treatment of assumptions about equality, including domain closure and the unique names hypothesis.

Finally, non-monotonic inference using partial model conjectures has been defined to always yield a consistent extension for a theory. For circumscription this is not the case, unless every model of the theory is an extension of a minimal model. In those instances where this is not the case, it has been shown that the circumscription formula can be inconsistent with an originally consistent theory (Etherington *et al.*, 1984).

## 3. HINTIKKA SETS AS PARTIAL MODELS

We now introduce a particular type of partial model, based on the method of analytic tableaux. [We do not give more than a cursory presentation of this method. See Smullyan (1968) for a general introduction; the method used here is based on work by Hintikka (1955).] Consider a theory *T*, which may be infinite. A tableau for *T* is a tree whose nodes are sentences, constructed in the following manner. The root of the tree is an arbitrary element of *T*. The tree is grown in a systematic manner from its leaves by adding new nodes, either elements of *T* or sentences derived from previous nodes by a small set of rules. Some of these rules, those dealing with disjunction, cause splits in the tree. The end result is a (perhaps infinite, but finitely branching) tree with an important property. All of the branches which are not *closed* (no sentence and its negation both appear on the branch) are partial models of $T$ of a certain sort. To take a very simple example, here is a tableau for the theory $T = \mathit{Bird}(\mathit{Tweety}) \vee \mathit{Bird}(\mathit{Opus})$:

$$\begin{tikzcd} & \mathit{Bird}(\mathit{Tweety}) \vee \mathit{Bird}(\mathit{Opus}) \arrow[dl, dash] \arrow[dr, dash] & \\ \mathit{Bird}(\mathit{Opus}) & & \mathit{Bird}(\mathit{Tweety}) \end{tikzcd}$$

There are two open branches, each comprising one partial model. The partial models are just sets of atoms and their negations. Our intended models, then, are *Herbrand interpretations* of the theory, in which all terms of the language denote themselves.

The method of analytic tableaux has two pleasing properties, which we state here without proof. The first is that a tableau for $T$ forms a complete survey of all of the Herbrand interpretations of $T$: any such interpretation is isomorphic to the extension of some open branch of the tableau. The second is that the partial models make the minimal assumptions about the equality of terms. This is a consequence of the particular effect of the rule that operates on existential statements. If a node in the tree has the form $\exists x . A[x]$, this rule allows the introduction of a node $A[a]$, where the constant $a$ is *new*, that is, has not been previously used in the tableau. So, in effect, this rule makes no assumption about the identity of the individual satisfying $A[x]$, since $a$ could be equal or not equal to any term already in the tableau.

### 3.1. Hintikka sets

We now give a formal definition of partial models based on tableaux. Let $L$ be a first-order language with equality and constant symbols, but no other function symbols. $A[\bar{x}]$ is a formula in which the free variables $\bar{x} = x_1, x_2, \ldots, x_n$ occur. $A[\bar{a}]$ is the result of substituting $a_i$ for each corresponding $x_i$. If $S$ is a set of formulas, the *universe* of $S$ is the set of constants of $S$.

The set of sentences on an open branch of a tableau is called a *Hintikka set.* A Hintikka set $S$ with universe $U$ satisfies the following conditions.

1. If $\neg A \in S$, then $A \notin S$.
2. If $A \wedge B \in S$, then both $A \in S$ and $B \in S$.
3. If $A \vee B \in S$, then either $A \in S$ or $B \in S$.
4. If $\exists x . A[x] \in S$, then for some constant $a \in U$, $A[a] \in S$.
5. If $\forall x . A[x] \in S$, then for all constants $a \in U$, $A[a] \in S$
6. $a \neq a \notin S$ for all $a \in U$.
7. If $a = b \in S$, then $A[a] \in S$ if and only if $A[b] \in S$.

In a Hintikka set, the equality predicate obeys the standard constraints of symmetry, reflexivity, transitivity, and substitution. Here are some examples of Hintikka sets that are open branches of tableaux, rooted in the first element of the set.

*Example* 3.1. $\{Pa \vee Pb, Pa\}$ with universe $\{a, b\}$.

*Example* 3.2. $\{Pa \wedge Pb \wedge \forall x.x = a,\ \ \forall x.x = a,\ \ Pa,\ \ Pb,\ \ a = a,\ \ b = a\}$ with universe $\{a, b\}$.

*Example* 3.3. $\{\exists x.\forall y.Rxy, \forall y.Ray, Raa\}$ with universe $\{a\}$.

*Example* 3.4. $\{\forall x.\exists y.Rxy,\ \exists y.Ra_1y,\ Ra_1a_2,\ \exists y.Ra_2y,\ Ra_2a_3,\ \exists y.Ra_3y,\ Ra_3a_4, \ldots\}$ with universe $\{a_1, a_2, \ldots\}$.

Example 3.4 has an infinite domain.

*Example* 3.5. Let $H$ be the conjunction of

1. $\forall x.\exists y.Rxy$
2. $\exists y.\forall x.\neg Rxy$
3. $\forall xyz.Rxy \wedge Rxz \supset y = z$
4. $\forall xyz.Ryz \wedge Rzx \supset y = z$

There is one Hintikka set $S$ of $T$, for which

$$v(S) = \bigcup_{i \geq 0} \{Ra_ia_{i+1}\} \cup \bigcup_{j \neq i+1} \{\neg Ra_ia_j\}$$

If we interpret the constant $a_i$ as the natural number $i$, this model is the natural numbers with $a_0 = 0$ being the least element.

In a Hintikka set $S$ with universe $U$, the atoms (atomic sentences) and their negations comprise a partial interpretation over the domain $U$, by specifying some of the positive and negative instances of relations. This interpretation is Herbrand in the sense that each constant in $U$ refers to itself. We call the set of atoms and their negations $v(S)$. $v(S)$ is *complete* if for every atom $p$ it contains either $p$ or $\neg p$.

A complete set $v(S)$ is called an *interpretation set with domain U*. It generates an interpretation by assigning the atom $Ra$ the value *true* if it is in $v(S)$, and *false* if its negation is. The truth value of compound statements is determined by the normal rules for quantifiers and Boolean operators. We write $v(S) \models T$ if every member of $T$ is assigned the value *true* (*i.e.* is satisfied) by $v(S)$, and call $v(S)$ a *model set for T*.

A *partial interpretation* is an atom set $v(S)$ that is not complete. It is an important property of a Hintikka set $S$ that $v(S)$ can always be extended, by the addition of atoms and their negations, to a complete Hintikka set; further, every such extension satisfies every sentence in $S$. This result is Hintikka's Lemma. If every extension of a partial interpretation satisfies a sentence $T$, we write $v(S) \models T$, and say that $v(S)$ is a *partial model* for $T$. Using this notation, we can write Hintikka's lemma as

$$v(S) \models S. \tag{3.1}$$

Example 3.2 has a complete atom set except for equality statements of symmetry and reflexivity (which we will generally ignore from now on, assuming they are always present); it is easy to verify that $v(S) \models S$. Example 3.3 is also complete. Example 3.1 has three extensions, one with $\{a = b, Pb\}$, one with $\{a \neq b, Pb\}$, and one with $\{a \neq b, \neg Pb\}$. Example 3.4 has an infinite number of extensions. Example 3.5 has only one extension, formed by including $\bigcup_{i \neq j} a_i \neq a_j$.

Finally, we want to make sure that the partial models contain the least information compatible with satisfying a theory. Consider the theory $T = Pa \vee (Pa \wedge Pb)$, which is equivalent to $Pa$. A tableau for $T$ generates two partial models, $\{Pa\}$ and $\{Pa, Pb\}$. However, the second partial model is an extension of the first. So we consider only those partial models of a theory that are *minimal*: there are no partial models which are proper subsets. We have not yet been able to prove that such minimal partial models always exist. However, the situation appears hopeful, because we are restricting our attention to Herbrand models generated by the tableau method. A case in point is the theory $T$ of Example 3.5, which has no minimal model in general, but has a single partial model in its tableau. This corresponds well with our intuitions, since all models of $T$ are isomorphic to the natural numbers.

## 4. NON-MONOTONIC INFERENCE ON PARTIAL MODELS

We consider four types of non-monotonic inference: the 'unique names' assumption, domain closure, disjoint domains, and default reasoning. All can be defined as conjectures about the *extension* of partial models, that is, we derive a theory $T'$ from $T$ by considering only a subset of the possible extensions of partial models for $T$.

By way of exposition, we first present a simple conjecture called *negative extension* (or *NE*) which is closely related to circumscription and the closed world assumption. Given a partial model, *NE* picks out that subset of its extensions which are maximal in negative atoms. Put another way, *NE* 'fills in' the missing information in a partial model by always adding negated atoms. There is an exception for equality: no equality atoms are added.

*Example* 4.1. Let $T = Pa \vee (Pb \wedge Pc)$; there are two partial models, $\{Pa\}$ and $\{Pb, Pc\}$. These give the following conjectured extensions under *NE*:

$$\begin{array}{l} \{Pa\} \\ \{Pa, \neg Pb\} \\ \{Pa, \neg Pb, \neg Pc\} \\ \{Pa, \neg Pb, \neg Pc, \neg Pe_1\} \\ \vdots \end{array} \tag{4.1}$$

and

$$\begin{aligned}&\{Pb, Pc\}\\ &\{Pb, Pc, \neg Pa\}\\ &\{Pb, Pc, \neg Pa, \neg Pe_1\}\\ &\vdots\end{aligned} \tag{4.2}$$

In the first of these, the sentence $\forall x\,.\, Px \supset x = a$ is satisfied; in the second, $\forall x\,.\, Px \supset (x = b \vee x = c)$. We have

$$T \vdash_{NE} T \wedge [(\forall x\,.\, Px \supset x = a) \vee (\forall x\,.\, Px \supset (x = b \vee x = c))] \tag{4.3}$$

or, equivalently,

$$(Pa \wedge \forall x\,.\, Px \supset x = a) \vee (Pb \wedge Pc \wedge \forall x\,.\, Px \supset (x = b \vee x = c)). \tag{4.4}$$

This result can be compared with the circumscription of $T$ (with respect to $P$), which yields:

$$\begin{aligned}&(Pa \wedge \forall x\,.\, Px \supset x = a) \vee\\ &\qquad (Pb \wedge Pc \wedge \forall x\,.\, Px \supset (x = b \vee x = c) \wedge a \neq b, c).\end{aligned} \tag{4.5}$$

The difference lies in the treatment of equality. In general, the conjecture *NE* will make fewer assumptions about the equality of terms than the corresponding circumscription.

### 4.1. The unique names hypothesis

This is the assumption that distinct names refer to distinct objects. The term was introduced by Reiter (1980) in formalizing a common naming convention in data bases. It is often useful to make this assumption in some form in common-sense reasoning. There are three additional criteria in this case:

1. It should be possible to state in the theory $T$ that some distinct terms are equal (e.g. *Morningstar* = *Eveningstar*).

2. It should also be possible to exclude terms from the unique names hypothesis by saying for a term that it may be equal to another term, without saying what that other term is. Skolem constants, for example, have this property.

3. The assumption of unique names should be defeasible, because it may turn out later that two distinct names actually do refer to the same individual.

It is very easy to implement the unique names assumption for partial models, because they are defined to minimize equality assumptions among names: it is the conjecture that adds as many equalities as possible to a partial model. More formally, we define $UN\colon\{a_1, a_2, \ldots\}$ as the conjecture that picks out those extensions of a partial model which are maximal in inequalities of the form $a_i \neq a_j$, $i \neq j$. This forces the names $a_1, a_2, \ldots$ to be maximally unique, i.e. two names $a_i$ and $a_j$ are considered to refer to different individuals unless the partial model forces them to be the same.

*Example* 4.2. Continuing Example 4.1, let $T = Pa \vee (Pb \wedge Pc)$. $T \vdash_{UN:\{a,b,c\}} a \neq b \wedge b \neq c \wedge a \neq c$.

*Example* 4.3. Let $T$ be as defined in Example 3.5. $T \vdash_{UN:\{a_1,\ldots\}} T$.

*Example* 4.4. Let $T$ be the sentence $ms = es \wedge x_0 \neq c$. In this example, $es$ and $ms$ are constants that refer to the same individual, while $x_0$ is a skolem constant, i.e. it may refer to some individual already named; however it cannot refer to $c$. We form the $UN$ conjecture for the constants $ms$, $es$, and $c$:

$$T \vdash_{UN:\{ms,es,c\}} T \wedge c \neq ms. \tag{4.6}$$

That is, $c$ is different from $ms$ (and $es$), and it is not known whether $x_0$ is the same as $ms$ or not. The conjecture that $c \neq ms$ is defeasible, because if it is later learned that $c = ms$, this can be added to $T$, and

$$T \wedge c = ms \vdash_{UN:\{ms,es,c\}} T \wedge c = ms. \tag{4.7}$$

This last example shows that the three criteria above for the unique names hypothesis are satisfied by the conjecture $UN$.

### 4.2. Domain closure

This term was originated by Reiter (1980) in the logical reconstruction of data base theory, as the assumption that only the individuals mentioned by the data base exist. If $a_1, a_2, \ldots$ are the constants of the data base, then the domain closure axiom is

$$\forall x . (x = a_1 \vee x = a_2 \vee \cdots). \tag{4.8}$$

This axiom is finite if the number of constants is finite. In data base theory, domain closure is usually invoked along with the unique names hypothesis, so that there are as many individuals as there are constants.

We will pursue a more semantically oriented form of domain closure, so that we can work with theories (such as $T = \forall x . \exists y . Rxy$) that have models containing individuals not explicitly named in the theory. Call $DC$ the assumption that *only those individuals exist that are minimally required to satisfy a theory.* This idea was first articulated by McCarthy (1980). Note that it is a stronger assumption than domain closure, because it picks out models that are minimal in their domains. For $T = Pa \vee Pb$, domain closure only implies $\forall x . (x = a \vee x = b)$, while the minimality requirement forces $(\forall x . x = a) \vee (\forall x . x = b)$, i.e. either everything is $a$, or it is $b$.

In terms of conjectures on partial models, $DC$ picks out the extensions of a partial model with the fewest individuals. For Hintikka sets, $DC$ is implemented by picking extensions $V$ of a partial model $v(S)$ with the following properties:

1. The universe of $V$ is the universe of $S$.
2. $V$ is maximal in positive equality atoms, i.e. there is no extension of $v(S)$ that contains more occurrences of positive equality atoms.

The first condition conjectures that all actual individuals have already been named in $S$ (this is Reiter's domain closure), and the second attempts to give as many names as possible a common interpretation.

*Example* 4.5. Let $S$ be the Hintikka set of Example 3.4. All equalities of the form $a_i = a_j$ can be consistently added, so $DC$ picks out the single extension

$$\left\{v(S) \cup \bigcup \{a_i = a_j\}\right\}. \tag{4.9}$$

This is the Hintikka set with the smallest number of individuals satisfying $\forall x . \exists y . Rxy$. We thus have

$$\forall x . \exists y . Rxy \vdash_{DC} \forall xy . x = y \tag{4.10}$$

which means that $DC$ conjectures a one-element domain.

As noted, in general $DC$ is a stronger assumption than domain closure. As this next example shows, $DC$ tries to identify different names with the same individual, thereby reducing the size of the domain; often this is not the desired result.

*Example* 4.6. Let $T$ be $Pb \wedge Pc$. Then

$$T \vdash_{DC} \forall x . x = c \wedge \forall x . x = b \wedge b = c. \tag{4.11}$$

In practice, it would be useful to couple $DC$ with a conjecture about the uniqueness of names, as is done in data base theory (see Reiter, 1980). We define the conjecture $UN{:}\bar{a}$; $DC$ as first taking the subset of complete extensions based on $UN{:}\bar{a}$, then further pruning these extensions by the $DC$ conjecture.

*Example* 4.7. Let $T$ be as in Example 4.6.

$$T \vdash_{UN:\{b,c\};DC} b \neq c \wedge \forall x . (x = b \vee x = c). \tag{4.12}$$

## 4.3. Disjoint domains

Often we wish to assume that individuals cannot belong to two different groups, e.g. one is normally either a Democrat or a Republican (or neither), but not both. However, we would like this assumption to be defeasible, since it could turn out that a person is registered for both parties, but in different states.

Let $P_1, \ldots, P_n$ be a set of monadic predicates that we assume to be disjoint. The disjoint domain axioms are expressed as:

$$\bigwedge_{i \neq j} \forall x . P_i x \supset \neg P_j x. \tag{4.13}$$

We cannot just add these axioms to a theory, however, because of the defeasibility condition. However, the disjoint domain assumption can be stated as the following conjecture: $DD:\{P_1 \cdots P_n\}$ picks out those complete extensions of a partial model with exactly one positive atom from the set $\{P_1a, \ldots, P_na\}$, for every constant $a$.

*Example* 4.8. Let $Dem\,x$ and $Rep\,x$ be predicates we wish to be disjoint, and let $\{Dem\,a\}$ be a partial model. $DD:\{Dem, Rep\}$ picks out the following extensions:

$$\begin{aligned}
&\{Dem\,a, \neg Rep\,a\}\\
&\{Dem\,a, \neg Rep\,a, Dem\,e_1, \neg Rep\,e_1\}\\
&\{Dem\,a, \neg Rep\,a, \neg Dem\,e_1, Rep\,e_1\}\\
&\vdots
\end{aligned} \tag{4.14}$$

and thus

$$\begin{aligned}
Dem\,a \vdash_{DD:\{Dem,Rep\}} Dem\,a &\wedge \neg Rep\,a\\
&\wedge \forall x\,.\,Dem\,x \supset \neg Rep\,x\\
&\wedge \forall x\,.\,Rep\,x \supset \neg Dem\,x.
\end{aligned} \tag{4.15}$$

The following theorem shows that $DD$ implies the disjoint domain axioms when there is no evidence to the contrary.

*Theorem* 4.1. Let $T$ be any tautology.

$$T \vdash_{DD:\{P_1 \cdots P_n\}} \bigwedge_{i \neq j} \forall x\,.\,P_ix \supset \neg P_jx \tag{4.16}$$

*Proof.* The single partial model of a tautology is the empty set. $DD:\{P_1 \cdots P_n\}$ will pick out all models in which $P_ia$ and $P_ja$ do not coexist, for all $a$ and $i \neq j$.

### 4.4. Default reasoning

Default reasoning is the assumption of propositions that have a reasonable chance of being true, given the available information. Formally, we implement defaults as conjectures on partial models, where the conjecture mentions specified and unspecified parts of the model. Consider, for example, the default rule that birds normally fly. Let $Bird\,x$ and $Fly\,x$ be the relevant predicates. We express the application of the default to a partial model $v(S)$ by the rule

If $Bird\,x$ is specified by $v(S)$ and $\neg Fly\,x$ is not specified by $v(S)$, then only consider those extensions of $v(S)$ that specify $Fly\,x$. (4.17)

We need a language for expressing default conjectures of this sort. A simple one can be formed by considering quantifier-free formulas, perhaps with free variables, together with the monadic operators $\Box$ and $\Diamond$. The default rule 4.17 would be expressed as

$$\alpha = \Box Bird\, x \wedge \Diamond Fly\, x \Rightarrow Fly\, x \tag{4.18}$$

$\Box p$ means that $p$ is true in *all* extensions of the partial model, and $\Diamond p$ that $p$ is true in *some* extension ($\Diamond = \neg\Box\neg$). If the formula to the left of the $\Rightarrow$ sign is satisfied by a partial model $v(S)$ for some instantiation $x = a$, then $\alpha$ picks out only those extensions of $v(S)$ containing $Fly\ a$, the instantiated expression to the right of $\Rightarrow$. Again, it is useful to think of $\alpha$ as 'filling in' a partial model that contains $Bird\, x$, by adding $Fly\, x$ if possible.

*Example* 4.9. This is from McCarthy (1984). There are ostriches, penguins, and canaries. Unless a bird is known to be an ostrich or penguin, we assume that it can fly. Let

$$T = \forall x\,.\, Ostrich\, x \supset \neg Fly\, x \wedge \forall x\,.\, Penguin\, x \supset \neg Fly\, x \tag{4.19}$$

and let $\alpha$ be the conjecture (4.18) above. We have

$$T \wedge Bird\ a \vdash_{\alpha} Fly\ a. \tag{4.20}$$

It is interesting to combine the default rule with assumptions about the disjointness of canaries, ostriches, and penguins:

$$T \wedge Canary\ a \vdash_{\alpha;DD:\{Ostrich,Penguin,Canary\}} Fly\ a \wedge \neg Ostrich\ a \wedge \neg Penguin\ a \tag{4.21}$$

and

$$T \wedge Ostrich\ a \vdash_{\alpha;DD:\{Ostrich,Penguin,Canary\}} \neg Fly\ a \wedge \neg Canary\ a \wedge \neg Penguin\ a \tag{4.22}$$

## 5. CONCLUSION: SOME ISSUES

### 5.1. Multiple conjectures

One of the pleasing aspects of a partial model approach is that conjectures of different sorts can be intermixed, as in the domain closure and unique names hypothesis of Example 4.7, and disjoint domain and default rules in Example 4.9. In both these examples there is an obvious order of application of the conjectures. Ordering is important because we first prune possible extensions with one conjecture, and then apply another conjecture to the result; doing this in a different order can lead to different sets of extensions. Ordering is a useful property when conjectures have readily defined priorities; however, especially in the case of default rules, conjectures may have roughly equal weights. Reiter (1980) gives the example the Republicans are normally non-pacifists and Quakers pacifists, so what about Richard Nixon, who is both a Quaker and Republican? As conjectures, these are

1. $\Box Rep\, x \wedge \Diamond\neg Pacifist\, x \Rightarrow \neg Pacifist\, x$.
2. $\Box Quaker\, x \wedge \Diamond Pacifist\, x \Rightarrow Pacifist\, x$.

If we apply (1) first, Nixon will be a non-pacifist; if (2), a pacifist. If both defaults are equal in their plausibility, it would be better not to conclude anything. We could try applying them in parallel, that is, taking the *intersection* of the extensions allowed by (1) and (2). However, this intersection would be empty in the present case, an undesirable result (and, by definition, not a conjecture). Instead, we might use the following rules:

1. $\Box Rep\, x \wedge \neg\Box Quaker\, x \wedge \Diamond\neg Pacifist\, x \Rightarrow \neg Pacifist\, x$.
2. $\Box Quaker\, x \wedge \neg\Box Rep\, x \wedge \Diamond Pacifist\, x \Rightarrow Pacifist\, x$.

But this is not entirely satisfactory either, because the modularity of the rules is compromised.

### 5.2. Proof theory

This is still unexplored. However there are some directions that appear promising.

1. Theories with finite tableaux. If a theory has a finite tableau, then it has a finite number of partial models, and it is possible to work directly with these. The chief syntactic class with this property are the $\exists\forall$-theories: those whose existential quantifiers all precede universals in prenex form.

2. Approximations based on one or a few partial models of a theory. All of the atoms (positive and negative) of the theory are kept as a partial model, while the more complicated axioms of the theory are treated procedurally as a means of deriving more atoms. Important disjunctions may be split into cases, producing more than one partial model. Note that this is the strategy of typical first-order AI knowledge bases (Nilsson, 1980). For syntactic classes that have a unique partial model, this is a complete technique. An interesting example here is provided by Horn-clause theories, for which the unique partial model is the intersection of all their Herbrand models.

**Acknowledgements**

I thank Benjamin Grosof and especially David Israel for enlightening discussions on this topic. This research was made possible in part by a gift from the System Development Foundation. It was also supported by Grant N00014-80-C-0296 from the Office of Naval Research.

## REFERENCES

Etherington, D. W., Mercer, R. E., and Reiter, R. (1984) On the adequacy of predicate circumscription for closed-world reasoning. *AAAI Workshop on Non-Monotonic Reasoning*. American Association for Artificial Intelligence, Menlo Park, Calif.

Gallaire, H., Minker, J., and Nicholas, J. M. (1978) An overview and introduction to logic and data bases. In *Logic and data bases* (eds H. Gallaire and J. Minker). Plenum Press, New York.

Hintikka, K. J. J. (1955) Form and content in quantification theory. *Acta Philosophica Fennica* **8**, 7–55.

Lifschitz, V. (1984) Some results on circumscription. *AAAI Workshop on Non-Monotonic Reasoning*. American Association for Artificial Intelligence, Menlo Park, Calif.

McCarthy, J. (1980) Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence* **13**, 1–2.

McCarthy, J. (1984) Applications of circumscription to formalizing common sense knowledge. *AAAI Workshop on Non-Monotonic Reasoning*. American Association for Artificial Intelligence, Menlo Park, Calif.

Minker, J. and Perlis, D. (1985) Completeness results for circumscription. University of Maryland, College Park, Md. Unpublished manuscript.

Nilsson, N. (1980) *Principles of artificial intelligence*. Tioga, Palo Alto, Calif.

Reiter, R. (1980) A logic for default reasoning. *Artificial Intelligence* **13**, 1,2.

Reiter, R. (1980) Equality and domain closure in first-order data bases. *J. Association for Computing Machinery* **27**(2), 235–49.

Smullyan, R. M. (1968) *First-order logic*. Springer-Verlag, New York.

2

# Equational Programming

N. Dershowitz and D. A. Plaisted*

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA

**Abstract**

Conditional (directed) equations provide a paradigm of computation that combines the clean syntax and semantics of both PROLOG-like logic programming and (first-order) LISP-like functional (applicative) programming in a uniform manner. For functional programming, equations are used as conditional rewrite rules; for logic programming, the same equations are employed for 'conditional narrowing'. Increased expressive power is obtained by combining both paradigms in one program.

## 1. INTRODUCTION

Equations can be used to compute by repeatedly substituting equal terms in a given formula, until the simplest possible form is obtained. Such a computation scheme is similar to that of (first-order) 'functional', or 'applicative', programming languages, such as LISP (McCarthy *et al.*, 1965) and its 'pattern-directed' derivatives. Programs can also be written as a set of equations between formulas or equivalences between statements in logic, and then executed by applying a special-purpose theorem prover that derives consequences from the given formulas until the desired output values are obtained. This latter form of computation is similar to the 'logic-programming' paradigm, as exemplified by PROLOG (Clocksin and Mellish, 1984). In this paper, we explore the possibility of using *conditional equations* to provide a uniform framework for combining PROLOG-like logic programming with LISP-like functional programming.

A *functional* (*applicative*) *program* is a set of *directed* equations used for computing. For example, a LISP-like program for concatenating two lists of elements where **nil** is the empty list and $\cdot$ denotes the function **cons** is shown below. The symbol $\Leftarrow$ has the declarative meaning 'is

* Now at the Department of Computer Science, University of North Carolina, USA.

*Functional list append*

$$\begin{aligned} append(U, V) \Leftarrow\ & \textbf{if } U = \textbf{nil} \\ & \textbf{then } V \\ & \textbf{else } \textbf{car}(U) \cdot append(\textbf{cdr}(U), V) \end{aligned}$$

equal', but operationally restricts the use of an equation to replacing instances of the left-hand side with the corresponding right-hand side. [Throughout this paper we follow the convention of using lower-case words for individual and function constants, lower-case letters are arbitrary terms or functions, upper-case for free (universally quantified) variables. Bold-face is used for standard built-in functions and constants; italics for user-defined ones.] Given the term

$$append(cho \cdot co \cdot \textbf{nil}, la \cdot te \cdot va \cdot \textbf{nil}),$$

the above program will compute the result of appending the two-element list $cho \cdot co \cdot \textbf{nil}$ [with parentheses, that should be $cho \cdot (co \cdot \textbf{nil})$] to the front of the list $la \cdot te \cdot va \cdot \textbf{nil}$. With the usual 'call-by-value' semantics, the *leftmost innermost* occurrence of a defined function symbol in a term is replaced by the *value* of the right-hand side of its definition. Furthermore, after evaluating the condition in an **if** . . . **then** . . . **else** . . . expression to either **true** or **false**, only the corresponding **then** or **else** branch is evaluated, not both.

*Pattern-directed* functional languages include SASL (Turner, 1979), HOPE (Burstall *et al.*, 1980), OBJ2 (Futatsugi *et al.*, 1985), ML (Gordon *et al.*, 1979), rewrite languages (O'Donnell, 1985), and PLANNER-like languages (Hewitt, 1971). In these languages, the left-hand side of an equation need not be of the restricted form $f(X_1, \ldots, X_n)$ where $f$ is a defined function and $X_1, \ldots, X_n$ are distinct variables. More than one equation may be given for $f$, though some restrictions on the form of left-hand sides are often imposed. For example, the following version of *append* has two equations for the two list *constructor* functions **nil** and **cons**:

*Pattern-directed append*

$$append(\textbf{nil}, V) \Leftarrow V$$
$$append(A \cdot U, V) \Leftarrow A \cdot append(U, V)$$

A *logic program,* as described in Kowalski (1974), is a set of Horn clauses used as a pattern-directed program that *searches* for output terms satisfying a given goal for given input terms. In this paradigm, the *append* program could be expressed as demonstrated below.

*Logic list append*

$$append(\mathbf{nil}, V, V)$$
$$append(A \cdot U, V, A \cdot W) \;:-\; append(U, V, W)$$

Here, the symbol :– has the declarative meaning 'is implied by'; operationally such an implication is used to replace a goal of the same form as the left-hand side with the corresponding right-hand side subgoal(s). Given a goal

$$append(cho \cdot co \cdot \mathbf{nil}, la \cdot te \cdot va \cdot \mathbf{nil}, Z),$$

the above program generates the subgoals

$$append(co \cdot \mathbf{nil}, la \cdot te \cdot va \cdot \mathbf{nil}, Y)$$
$$append(\mathbf{nil}, la \cdot te \cdot va \cdot \mathbf{nil}, X),$$

and then returns the answer $Z = cho \cdot Y = cho \cdot co \cdot X = cho \cdot co \cdot la \cdot te \cdot va \cdot \mathbf{nil}$. In PROLOG, 'SLD-resolution' is used to always solve subgoals from left to right. Furthermore, applicable clauses are attempted in the order of appearance in the program.

In previous research (Dershowitz, 1983, 1984; Dershowitz and Josephson, 1984), we investigated the use of unconditional rewrite systems for logic programming. A *rewrite system* (see Huet and Oppen, 1980) is a set of *directed* equations (or equivalences) used as a *non-deterministic* pattern-directed program that returns as output a simplified term equal to a given input term. An example is given below using a four-rule rewrite system for *append*.

*List append*

$$\begin{aligned} append(\mathbf{nil}, V) &\rightarrow V \\ append(U, \mathbf{nil}) &\rightarrow U \\ append(A \cdot U, V) &\rightarrow A \cdot append(U, V) \\ append(append(U, V), W) &\rightarrow append(U, append(V, W)) \end{aligned}$$

Rules may be applied in any order to any matching subterm until no further applications are possible; the order of rules is immaterial. Thus applying the rules to the term

$$append(va \cdot \mathbf{nil}, \mathbf{nil}),$$

one gets either the rewrite sequence

$$append(va \cdot \mathbf{nil}, \mathbf{nil}) \Rightarrow va \cdot append(\mathbf{nil}, \mathbf{nil}) \Rightarrow va \cdot \mathbf{nil},$$

or, using only the second rule,

$$append(va \cdot \mathbf{nil}, \mathbf{nil}) \Rightarrow va \cdot \mathbf{nil}.$$

The same rules are also used to solve goals by a process called 'narrowing'. If the left-hand side of a rule unifies with any subterm of a goal, then the goal is *narrowed* by applying the unifying substitution to the goal and then applying the rule to rewrite the subterm.

Conditional equations provide a coherent means for combining functional programming with logic programming. A *conditional equation* is a formula of the form

$$p \supset l = r,$$

meaning that the term $l$ is equal to the term $r$ when the condition $p$ holds. In general, there may be variables $X$, $Y$, etc. in $p$, $l$ and/or $r$, in which case the conditional equation is meant to hold for *all* terms $X$, $Y$, etc. If any term containing an instance of $l$ is 'less defined' than the corresponding term with $r$ in place of $l$, then the conditional equation may be *directed.* (See Section 5.1.) A directed equation is used to 'substitute equals for equals' only from left to right, and we write it as a *conditional rule*:

$$l :- p \rightarrow r.$$

Either (or both) of the rule parts $:-p$ and $\rightarrow r$ may be omitted, in which case it is taken to the **true**. (Variables on the right-hand side should also appear in one of the other two parts, and no left-hand side should be the term **true**.) A rule with only a left-hand side $l$ is called as *assertion*; one with no condition $p$ is a *rewrite rule*; one with no right-hand side $r$ is a *logic rule.*

A (*conditional*) *rewrite program* is a system of such conditional, directed equations. Each equation may be used in two distinct ways: it can be used to *simplify* a subterm that matches its left-hand side, and it can be used to *narrow* a subterm that unifies with its left-hand side. Thus, a rewrite system can be used to compute by repeatedly substituting equal terms in a given term, until the simplest form possible is obtained. A system can also be used to compute by deriving consequences of given equations until the desired output values are obtained. As we will see, simplifications are irrevocable, while narrowings are provisional.

In the next section we consider functional programming using equations for simplification and rewriting, and in the section that follows it we consider logic programming using equations for unification and narrowing. Section 4 shows some of the benefits that can be obtained by combining both programming forms in one program. Correctness issues are addressed in Section 5 and implementation issues in Section 6. They are followed by a comparison with related work.

## 2. FUNCTIONAL PROGRAMMING

A (*conditional*) *rewrite system* $R$ is a finite set of rewrite rules, each of the form

$$l[\bar{X}] :- p[\bar{X}] \rightarrow r[\bar{X}],$$

where $l$ and $r$ are terms and $p$ is a predicate. Terms in general contain variables (these are the $\bar{X}$), but the *right-hand side* $r$ and the *condition* $p$ should only contain variables that appear in the *left-hand side* $l$. (This restriction will be relaxed in the next section.) Such a rule may be applied to a term $t$ if a subterm $s$ of $t$ *matches* (by 'one-sided' unification) the left-hand side $l$ with some substitution $\sigma$ of terms for the variables in $l$, and if the corresponding condition $p\sigma$ is true, where $p\sigma$ denotes the term $p$ after making the substitution $\sigma$ for its variables. The rule is applied by replacing the subterm $s = l\sigma$ in $t$ with the right-hand side $r\sigma$. The choice of which rule to apply where is made non-deterministically from among all possibilities. We write $t \Rightarrow t'$ to indicate that a term $t'$ is *derivable* from the term $t$ by a single application of some rule in $R$. When we said above that $p\sigma$ must be true for the rule to be applied, we meant $p\sigma \Rightarrow \cdots \Rightarrow$ **true**, i.e. that $p\sigma$ *reduces* to the constant **true** via zero or more rule applications. There is no backtracking over reductions. (There is backtracking over deductions, as we will see in the next section.)

A functional program definition of the form

$$f(\bar{X}) \Leftarrow \textbf{if } p[\bar{X}] \textbf{ then } r[\bar{X}] \textbf{ else } s[\bar{X}]$$

can be translated into an unconditional system

$$\begin{aligned} f(\bar{X}) &\rightarrow f'(p[\bar{X}], \bar{X}) \\ f'(\textbf{true}, \bar{X}) &\rightarrow r[\bar{X}] \\ f'(\textbf{false}, \bar{X}) &\rightarrow s[\bar{X}], \end{aligned}$$

where $f'$ is a new function symbol. This has the effect of ensuring that the condition is evaluated only once before either branch is explored. If the condition $p[\bar{X}]$ evaluates to **true** for the given values of $\bar{X}$, the term $f(\bar{X})$ gets eventually replaced by $r[\bar{X}]$; if $p[\bar{X}]$ evaluates of **false**, then $f(\bar{X})$ is replaced by $s[\bar{X}]$. The one-line rule

$$f(\bar{X}) \rightarrow \mathbf{if}\ p[\bar{X}]\ \mathbf{then}\ r[\bar{X}]\ \mathbf{else}\ s[\bar{X}]$$

may be considered, then, as an abbreviation for the above three rules.

An alternative rewrite program for conditional expressions, using conditional rules and no new symbols, would be

$$\begin{aligned} &f(\bar{X}) :- p[\bar{X}] &&\rightarrow r[\bar{X}] \\ &f(\bar{X}) :- \mathbf{not}(p[\bar{X}]) &&\rightarrow s[\bar{X}], \end{aligned}$$

where **not(false)** evaluates to **true**. Often, conditions may instead be expressed as left-hand side patterns. Consider, for example, the following program for computing the union of two sets (of numbers, say) represented as lists without repetitions. That is, given two lists $X$ and $Y$, it returns a list $union(X, Y)$, containing those elements that appear in at least one of the input lists, as shown below.

*List union*

| | | |
|---|---|---|
| $union(\mathbf{nil}, Y)$ | | $\rightarrow Y$ |
| $union(X, \mathbf{nil})$ | | $\rightarrow X$ |
| $union(A \cdot X, Y)$ | $:- member(A, Y)$ | $\rightarrow union(X, Y)$ |
| $union(A \cdot X, Y)$ | $:- \mathbf{not}(member(A, Y))$ | $\rightarrow A \cdot union(X, Y)$ |
| $member(A, \mathbf{nil})$ | | $\rightarrow \mathbf{false}$ |
| $member(A, A \cdot Y)$ | | |
| $member(A, B \cdot Y)$ | $:- \mathbf{not}(A = B)$ | $\rightarrow member(A, Y)$ |
| $I = J$ | $:- \mathbf{number}(I, J)$ | $\rightarrow \mathbf{eq}(I, J)$ |

Here *union* is the function being defined, *member* is an auxiliary predicate testing for membership of an element in a list, **eq** is a built-in predicate that tests for equality of numbers, and **number** is a built-in predicate that returns **true** if all its argument are numbers (and **false** otherwise). In place of the logic rule

$$member(A, Z) :- A = \mathbf{car}(Z),$$

this program has an assertion

$$member(A, A \cdot Y)$$

with the pattern $A \cdot Y$. The condition **number** is necessary in

$$I = J :- \mathbf{number}(I, J) \rightarrow \mathbf{eq}(I, J),$$

since **eq** is only intended to work for built-in data types. Note that we must have the **false** case for *member* for the fourth *union* rule to work, since negation is not being handled 'by failure' (cf. Clark, 1978).

## 3. LOGIC PROGRAMMING

Rewrite systems may be used as 'logic programs' (Kowalski, 1974), in addition to their straightforward use for computation by rewriting, illustrated in the previous section. The programming paradigm described below allows for the advantageous combination of both computing modes. The result is a PROLOG-like programming language, the main difference being that rewrite rules are conditional equivalences, rather than implications in Horn-clause form. Any (pure) PROLOG statement may be directly translated into a rewrite rule: the clause

$$l :- p, r$$

corresponds to the identical rule. A rule of the form

$$l :- p \rightarrow r$$

is stronger than the above Horn clause and means that $p \supset (l \equiv r)$. A rule like

$$l \rightarrow p \,\&\, r$$

is even stronger; it has $l$ true *if, and only if*, $p$ and $r$ both hold.

The next example given is a program $div(a, b, Q, R)$ to compute the quotient $Q$ and remainder $R$ of non-negative integer $a$ and positive integer $b$.

*Integer division*

| | | | |
|---|---|---|---|
| $div(X, Y, Q+1, R)$ | $:- X \geq Y$ | $\rightarrow$ | $div(X-Y, Y, Q, R)$ |
| $div(X, Y, 0, X)$ | $:- Y > X$ | | |
| $div(X, Y, 0, R)$ | $:- X \geq Y$ | $\rightarrow$ | **false** |
| $I > J$ | $:-$ **number**$(I, J)$ | $\rightarrow$ | **greater**$(I, J)$ |
| $I \geq J$ | $:-$ **number**$(I, J)$ | $\rightarrow$ | **not**(**less**$(J, I)$) |
| $I - J$ | $:-$ **number**$(I, J)$ | $\rightarrow$ | **diff**$(I, J)$ |

The first rule is the recursive case; the second is the base case; the third covers false cases; the remainder apply built-in functions to numbers.

The *resolution procedure* (Robinson, 1963) derives consequences of logical formulas written in 'clausal' form. Resolving a clause of the form

$$p \vee q$$

with another clause

$\neg r \vee s,$

when $p$ and $r$ are unifiable by $\sigma$, results in the new clause

$q\sigma \vee s\sigma.$

The *completion procedure* (Knuth and Bendix, 1970) was introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given unconditional equational theories. More recently, it has been applied to other aspects of equational reasoning (see, for example, Dershowitz, 1982), and in Dershowitz and Josephson (1984), Durand (1984), and Réty *et al.* (1985) it has been applied to logic programming, as well. Completion is, in a sense, an extension of resolution in that it allows unification at subterms. That is, a rule

$l \rightarrow r$

may be *overlapped* on a rule of the form

$u[s] \rightarrow t,$

whose left-hand side $u$ contains a subterm $s$ that is unifiable with $l$ via substitution $\sigma$ (the variables in the two rules are treated as disjoint). The result is one of the two rules:

$u[r]\sigma \rightarrow t\sigma$

or

$t\sigma \rightarrow u[r]\sigma$

where $u[r]\sigma$ is $u$ with its subterm $s$ replaced by $r$ and $\sigma$ applied. Which orientation is chosen depends on a well-founded ordering $>$ supplied to the procedure. If $u[r]\sigma > t\sigma$ in that ordering, the former is chosen; if $t\sigma > u[r]\sigma$, the latter is. (If the ordering is partial, it may be that neither orientation works.) *Narrowing* (Slagle, 1974) is a 'linear' restriction on completion, analogous to 'linear input resolution'. That is, program rules $l \rightarrow r$ are only overlapped on goal rules, not program rules on program rules, nor goal rules on goal rules. The orientation of subgoal rules is fixed, so no ordering is needed.

Formally, a *rewrite program* is a set of rewrite rules of the form

$l[\bar{X}] :- p[\bar{X}, \bar{Y}] \rightarrow r[\bar{X}, \bar{Y}],$

where the condition may contain variables $\bar{Y}$ not also on the left-hand side. To execute a rewrite program, we must adapt the narrowing process to rules with conditions; we call this adaptation *conditional narrowing*. In the next few paragraphs, we describe the details of rewrite program interpretation.

To begin a computation with a rewrite program, a *goal rule* is added to the system. Goal rules are of the form

$$g[\bar{x}, \bar{Z}] \rightarrow answer(\bar{Z}),$$

where $g$ is the *calling term* containing input values (i.e. irreducible variable-free terms) $\bar{x}$ and output variables $\bar{Z}$, and *answer* is the predicate symbol that will store the result. PROLOG goals of the form

$$:-\ q, r$$

correspond to goal rules

$$\mathbf{true} :-\ q, r \rightarrow answer(\bar{Z}),$$

where $\bar{Z}$ are the variables in $q$ and $r$. At each point in the computation, the current subgoal is of the general form

$$h :-\ q_1, \ldots, q_n \rightarrow answer(\bar{s}),$$

meaning that the answer is $\bar{s}$ if the subgoals $q_1, \ldots, q_n$, and $h$ are achieved (in that order). Given such a subgoal, and a rule

$$l :- p \rightarrow r$$

whose left-hand side $l$ can be unified with a non-variable subterm of $q_1$ via most general unifier $\sigma$, i.e. $q_1\sigma$ which contains $l\sigma$ when the goal and rule variables are treated as disjoint, the subgoal is *conditionally narrowed* to

$$h\sigma :-\ p\sigma, q_1\sigma[r\sigma], q_2\sigma, \ldots, q_n\sigma \rightarrow answer(\bar{s}\sigma).$$

At each such step, all possible simplifications (as in the previous section) are applied throughout. That is, if a left-hand side *matches* any subterm of a subgoal, that subterm is reduced. (Recall that for a conditional rule to apply, the condition must reduce to **true**. If the condition reduces to anything else, the rule is not applied. If it reduces to a term containing variables, rather than to **true** or **false**, then the potential simplification needs to be considered as a possible narrowing, since the reduced condition must first be narrowed to **true**.) Simplifying gives a new subgoal

$$h' :-\ p', q_1', q_2', \ldots, q_n' \rightarrow answer(\bar{s}).$$

where $h'$, $p'$, $q_1'$, etc. are all irreducible. If any of these is reduced to the term **false**, then the whole subgoal should be abandoned. (If, as can conceivably happen, the simplified condition $p'$ is just a Boolean variable $X$, then **true** needs to be substituted for $X$ throughout.) Only when all the conditions become **true**, and the subgoal is of the unconditional form

$$h' \rightarrow answer(\bar{s}),$$

are narrowing unifications attempted within $h'$, thereby possibly introducing new conditions. Computation ends when a *solution rule*

$$\mathbf{true} \rightarrow answer(\bar{u})$$

is generated, giving an answer $\bar{u}$ such that

$$g[\bar{x}, \bar{u}]$$

holds. Since in general, there may be many ways to achieve a subgoal, alternative narrowing computations must be attempted, either in parallel (until one succeeds) or sequentially (by backtracking upon failure).

Conditions, when separated by commas, are executed from left to right, and must all be true before the left-hand side is replaced by the right-hand side. Conditions separated by the symbol &, on the other hand, may be executed in any order. Such commas are just 'syntactic sugar' in that a rule

$$l :- p, q \rightarrow r$$

can always be replaced by a rule

$$l :- p \rightarrow if(q, r),$$

along with the general rule

$$if(Q, R) :- Q \rightarrow R,$$

each having only one condition.

To compute, for example, the quotient and remainder of two non-negative numbers $a$ and $b$ with the above program, the rule

$$div(a, b, Q, R) \rightarrow answer(Q, R)$$

is added, meaning that $Q$ and $R$ are the answer if and only if they are the quotient and remainder, respectively, of $a$ and $b$. The interpreter then generates a rule

$$\mathbf{true} \rightarrow answer(c, d),$$

containing the answer values $c$ and $d$ for $Q$ and $R$. For example, to compute the quotient and remainder of 7 and 3, the rule

$$div(7, 3, Q, R) \rightarrow answer(Q, R)$$

is added. Narrowing generates

$$div(7-3, 3, U, R) \rightarrow answer(U+1, R),$$

by applying the first program rule, which simplifies to

$$div(4, 3, U, R) \rightarrow answer(U+1, R),$$

applying the last rule for built-in subtraction. Using the first rule again gives

$$div(1, 3, V, R) \rightarrow answer(V+1+1, R).$$

Now the second rule yields the answer

$$\mathbf{true} \rightarrow answer(0+1+1, 1).$$

Note that this program can test whether *or not* two numbers have the given quotient and remainder. It is not, however, in a form that would allow computing the first argument, say, from the other three, unless the built-in **number**($I$, $J$) generates all instances of $I$ and $J$ that are numbers. A goal like

$$div(X, 3, 2, 1) \rightarrow answer(X)$$

generates the subgoal

$$div(X-3, 3, 2, 1) :- X \geq 3 \rightarrow answer(X),$$

but we gave no rules for reducing $X-3$ or solving $X \geq 3$ when $X$ is not a number satisfying **number**($X$, 3). See Section 6.1.

## 4. FEATURES

The two paradigms of computation illustrated in the preceding sections, viz. simplification and narrowing, can be combined in a single rewrite program. Every narrowing step is followed by as much simplification as possible. Simplification steps employ pattern matching, while narrowing involves unification. Simplifications are irrevocable, while narrowing steps are subject to backtracking.

As an example of the utility of applying simplifications at more than one level of a goal, consider the *list generator* situation.

*List generator*

| | |
|---|---|
| $listp(\mathbf{nil})$ | |
| $listp(A \cdot Y)$ | $\rightarrow listp(Y)$ |
| $length(\mathbf{nil})$ | $\rightarrow 0$ |
| $length(A \cdot Y)$ | $\rightarrow length(Y)+1$ |
| $length(Y)<0$ | $\rightarrow \mathbf{false}$ |
| $I<J :- \mathbf{number}(I, J)$ | $\rightarrow \mathbf{less}(I, J)$ |
| $I+1<J :- \mathbf{number}(J)$ | $\rightarrow I<\mathbf{sub1}(J)$ |
| $P \ \& \ \mathbf{false}$ | $\rightarrow \mathbf{false}$ |

A subgoal *listp*$(Z)$ can, by repeating the second rule, generate arbitrarily large lists $Z = A_1 \cdot A_2 \cdots A_n \cdot \mathbf{nil}$. But when combined with a test for length, as in

$$\mathit{listp}(Z) \;\&\; \mathit{length}(Z) < 10,$$

it will *reduce* to **false** after 10 narrowings (because at that point the second subgoal becomes **false**), thereby pruning an otherwise potentially infinite computation path.

**4.1. Solving equations**

The functional *union* program in Section 2 can be used, for example, in a logic program to find a list $Z$ and element $A$ such that

$$\{1\} \cup Z = \{A, 1\} \cup \{2, 3\}.$$

That goal can be expressed as

$$\mathit{union}(1 \cdot \mathbf{nil}, Z) = \mathit{union}(A \cdot 1 \cdot \mathbf{nil}, 2 \cdot 3 \cdot \mathbf{nil}) \;\&\; \mathbf{not}(A = 1) \rightarrow \mathit{answer}(A, Z),$$

where the condition $\mathbf{not}(A = 1)$ is needed to ensure that the list $A \cdot 1 \cdot \mathbf{nil}$ is a proper encoding of a set. To solve equations, we will also need an additional assertion

$$U = U.$$

The computation could then proceed as follows: beginning with the goal

$$\mathit{union}(1 \cdot \mathbf{nil}, Z) = \mathit{union}(A \cdot 1 \cdot \mathbf{nil}, 2 \cdot 3 \cdot \mathbf{nil}) \;\&\; \mathbf{not}(A = 1) \rightarrow \mathit{answer}(A, Z),$$

the third rule for *union* is applied, giving

$$\mathit{union}(1 \cdot \mathbf{nil}, Z) = 1 \cdot 2 \cdot 3 \cdot \mathbf{nil} \;\&\; \mathbf{not}(A = 1) :- \mathit{member}(A, 2 \cdot 3 \cdot \mathbf{nil}) \rightarrow \mathit{answer}(A, Z).$$

(The first rule for *union* cannot be used, since **nil** does not unify with $1 \cdot \mathbf{nil}$; the second rule makes $Z = \mathbf{nil}$, but then the equality fails; the fourth rule leads to other solutions.) Now the condition *member*$(A, 2 \cdot 3 \cdot \mathbf{nil})$ needs to be solved before anything else. One way to solve it is by letting $A = 2$, yielding

$$\mathit{union}(1 \cdot \mathbf{nil}, Z) = 1 \cdot 2 \cdot 3 \cdot \mathbf{nil} \rightarrow \mathit{answer}(2, Z)$$

after simplification. Using the third rule again, gives

$$Z = 1 \cdot 2 \cdot 3 \cdot \mathbf{nil} :- \mathit{member}(1, Z) \rightarrow \mathit{answer}(2, Z).$$

If $Z = 1 \cdot Y$, the condition is satisfied and it remains to solve

$$1 \cdot Y = 1 \cdot 2 \cdot 3 \cdot \mathbf{nil} \rightarrow \mathit{answer}(2, 1 \cdot Y).$$

Unifying the two sides of the equality, using the assertion $U = U$, solves the original goal:

$$\mathbf{true} \rightarrow \mathit{answer}(2, 1 \cdot 2 \cdot 3 \cdot \mathbf{nil}).$$

This computation yields as an answer, $A = 2$ and $Z = 1 \cdot 2 \cdot 3 \cdot \mathbf{nil}$, one solution out of many.

The *union* program can also be extended with rules like

$$\mathit{member}(A, \mathit{union}(X, Y)) \rightarrow \mathit{member}(A, X) \vee \mathit{member}(A, Y)$$

to help find, say, an $X$ such that $A$ is (or is not) a member of $\mathit{union}(X, Y)$, given $A$ and $Y$.

### 4.2. Assignment

The conditional part can be used for generalized assignment (subsuming **setq** in LISP and **is** in PROLOG) as in the *insertion sort* program.

*Insertion sort*

$$\begin{array}{rll}
\mathit{sort}(\mathbf{nil}) & & \rightarrow \mathbf{nil} \\
\mathit{sort}(A \cdot \mathbf{nil}) & & \rightarrow A \cdot \mathbf{nil} \\
\mathit{sort}(A \cdot Y) :- & Z \doteq \mathit{sort}(Y) & \rightarrow \mathit{insert}(A, Z) \\
\mathit{insert}(A, \mathbf{nil}) & & \rightarrow A \cdot \mathbf{nil} \\
\mathit{insert}(A, B \cdot Z) :- & \mathbf{not}(\mathbf{greater}(A, B)) & \rightarrow A \cdot B \cdot Z \\
\mathit{insert}(A, B \cdot Z) :- & \mathbf{not}(\mathbf{less}(A, B)) & \rightarrow B \cdot \mathit{insert}(A, Z) \\
\mathbf{nil} \doteq \mathbf{nil} & & \\
A \cdot Z \doteq A \cdot Z & & \\
\mathbf{nil} \doteq A \cdot Z & & \rightarrow \mathbf{false} \\
A \cdot Z \doteq \mathbf{nil} & & \rightarrow \mathbf{false}
\end{array}$$

The purpose of the condition $Z \doteq \mathit{sort}(Y)$ is to assign the *sorted* list to $Z$. Only when $\mathit{sort}(Y)$ is partially evaluated to a list of the form $A \cdot Y$ can the rules for $\doteq$ be applied; the term $\mathit{sort}(Y)$ itself cannot be assigned to $Z$, as would be the case were $=$ used. This has an effect similar to that of the 'read-only' function in Concurrent PROLOG (Shapiro, 1983). In general, one would want to have built-in assignment rules of this form for each built-in data type.

### 4.3. Functionality

The two main uses of 'cuts' in PROLOG are to avoid backtracking in the presence of 'functional dependencies' and to handle negation (see Clocksin and Mellish, 1984). With rewrite rules, functions can be handled directly by simplification which does not allow for backtracking. Thus, a goal of the form

$$p(f(t), Z) \rightarrow \mathit{answer}(Z),$$

for some term $t$, will first have $f(t)$ simplified, before continuing with the subgoal $p$. If $p$ then fails, no attempt is made to undo the computation of $f$. In particular, a goal of the form

$$Z = f(t) \rightarrow \mathit{answer}(Z)$$

makes the functional dependency explicit, and leaves no room for backtracking. After evaluating $f(t)$, only the equality axiom

$$U = U$$

applies.

### 4.4. Negation

The second use of 'cuts', negation, can be handled by evaluating false cases as well as true ones. For example, the program shown below computes the first prime numbers up to (but not including) $N$.

*Prime number generator*

| | | |
|---|---|---|
| $\mathit{prime}(N)$ | | $\rightarrow \mathit{sift}(\mathit{integers}(2, N))$ |
| $\mathit{integers}(K, N)$ | $:-$ $\mathbf{less}(K, N)$ | $\rightarrow K \cdot \mathit{integers}(K+1, N)$ |
| $\mathit{integers}(K, N)$ | $:-$ $\mathbf{not}(\mathbf{less}(K, N))$ | $\rightarrow \mathbf{nil}$ |
| $\mathit{sift}(N \cdot L)$ | | $\rightarrow N \cdot \mathit{sift}(\mathit{filter}(N, L))$ |
| $\mathit{sift}(\mathbf{nil})$ | | $\rightarrow \mathbf{nil}$ |
| $\mathit{filter}(M, \mathbf{nil})$ | | $\rightarrow \mathbf{nil}$ |
| $\mathit{filter}(M, N \cdot L)$ | $:-$ $\mathit{divides}(M, N)$ | $\rightarrow \mathit{filter}(M, L)$ |
| $\mathit{filter}(M, N \cdot L)$ | $:-$ $\mathbf{not}(\mathit{divides}(M, N))$ | $\rightarrow N \cdot \mathit{filter}(M, L)$ |
| $\mathit{divides}(M, N)$ | $:-$ $\mathit{div}(N, M, Q, 0)$ | |

Here *div* is the division program given earlier. Since $\mathit{divides}(M, N)$ returns **false** when $M$ does not divide $N$, the program can test for $\mathbf{not}(\mathit{divides}(M, N))$ without recourse to a new predicate 'not-divides', an additional argument to *divides*, or to 'negation by failure'.

Any 'closed-world' assumption can be made explicit, using the 'if-and-only-if' meaning of $\rightarrow$, as in the next example.

*Adam and Eve*

$$\begin{array}{rll}
\mathit{female}(X) & :-\ \mathit{person}(X) & \rightarrow \neg \mathit{male}(X) \\
\mathit{male}(X) & :-\ \mathit{person}(X) & \rightarrow X = \mathit{adam} \\
\mathit{person}(\mathit{adam}) & & \\
\mathit{person}(\mathit{eve}) & & \\
X = Y & :-\ \mathit{person}(X)\ \&\ \mathit{person}(Y) & \rightarrow \mathbf{eq}(X, Y) \\
\neg \mathbf{true} & & \rightarrow \mathbf{false} \\
\neg \mathbf{false} & &
\end{array}$$

Here, ¬ is *not* the built-in negation function and **eq** works for people, too. The goal *female*(*Y*) results in the following computation:

$$\begin{array}{rl}
\mathit{female}(Y) & \rightarrow \mathit{answer}(Y) \\
\neg \mathit{male}(Y) :-\ \mathit{person}(Y) & \rightarrow \mathit{answer}(Y) \\
\mathbf{true} & \rightarrow \mathit{answer}(\mathit{eve})
\end{array}$$

This works since it is explicitly given that a person is female if and only if she is non-male, and that *adam* is the only male in the (primeval) world. The first step replaces *female*(*Y*) with ¬*male*(*Y*). Then the condition *person*(*Y*) is solved. Letting $Y = \mathit{adam}$ fails, since ¬*male*(*adam*) reduces to **false**; letting $Y = \mathit{eve}$ succeeds.

It is this ability to express negation, and say that some goal is **false**, which allows for the pruning of fruitless paths. The *slow sort* example illustrates this.

*Slow sort*

$$\begin{array}{rll}
\mathit{sort}(\mathbf{nil}) & & \rightarrow \mathbf{nil} \\
\mathit{sort}(Y) & :-\ \mathit{perm}(Y, Z),\ \mathit{ordered}(Z) & \rightarrow Z \\
\mathit{ordered}(\mathbf{nil}) & & \\
\mathit{ordered}(A \cdot \mathbf{nil}) & & \\
\mathit{ordered}(A \cdot B \cdot Z) & :-\ \mathbf{not}(\mathbf{less}(B, A)) & \rightarrow \mathit{ordered}(B \cdot Z) \\
\mathit{ordered}(A \cdot B \cdot Z) & :-\ \mathbf{less}(B, A) & \rightarrow \mathbf{false} \\
\mathit{perm}(Y, B \cdot Z) & :-\ \mathit{append}(U, B \cdot V) \doteq Y & \rightarrow \\
 & & \quad \mathit{perm}(\mathit{append}(U, V), Z) \\
A \cdot Y \doteq B \cdot Z & & \rightarrow A = B\ \&\ Y \doteq Z \\
A \cdot Y \doteq \mathbf{nil} & & \rightarrow \mathbf{false}
\end{array}$$

It uses the previous program for *append*. Any permutation being generated by *perm* is pruned by *ordered* as soon as it contains two inverted elements. The definition for $\doteq$ prunes impossibly long instances of $append(U, B \cdot V)$.

A more general logical facility than negation that works for all Boolean combinations of predicates is provided by the rewrite system given below.

*Propositional calculus*

$$
\begin{aligned}
\neg U &\rightarrow U \equiv \mathbf{false}\\
U \vee V &\rightarrow (U\&V) \equiv U \equiv V\\
U \supset V &\rightarrow (U\&V) \equiv U\\
U \mathbin{\&} \mathbf{true} &\rightarrow U\\
U \mathbin{\&} \mathbf{false} &\rightarrow \mathbf{false}\\
U \mathbin{\&} U &\rightarrow U\\
U \equiv \mathbf{true} &\rightarrow \mathrm{U}\\
U \equiv U &\\
(U \equiv V) \mathbin{\&} W &\rightarrow ((U\&W) \equiv (V\&W)) \equiv W
\end{aligned}
$$

Using these rules requires associative–commutative unification (Stickel, 1981; Fages, 1984) for & and ≡ (*equivalence*). The advantage is that any propositional formula has a *unique* irreducible form. Propositionally valid formulae reduce to **true**; propositionally unsatisfiable ones reduce to **false** (Hsiang and Dershowitz, 1983).

Note that negation, combined with simplification, makes the ordering of subgoals less crucial, since pruning can be used to guarantee termination (Fribourg, 1985).

### 4.5. Streams

The use of 'streams' is illustrated in the modification of the prime number program shown on the next page. Invoking

$$prefix(Z, sift(integers(2))) \rightarrow answer(Z)$$

generates arbitrarily long sequences of prime numbers. Notice how new (primed) function names (e.g. *integers'*) are used for otherwise infinite 'streams'; a 'call-by-need' effect is obtained with the last set of rules for reinstating the original (unprimed) function names (cf. Tamaki and Sato, 1983). Other potentially non-terminating function definitions of the form $f(s) \rightarrow g(f(t))$ can always be systematically replaced by clauses of the form $f(s, g(u), N+1) :- f(t, u, N)$ in which an additional (artificial) argument bounds recursion depth (and may be solved for by narrowing).

*Prime number stream*

| | | |
|---|---|---|
| $prefix(N \cdot K, N \cdot L)$ | | $\rightarrow prefix(K, L)$ |
| $prefix(\mathbf{nil}, L)$ | | |
| $integers(K)$ | | $\rightarrow K \cdot integers'(K+1)$ |
| $sift(N \cdot L)$ | | $\rightarrow N \cdot sift'(filter(N, L))$ |
| $filter(M, N \cdot L)$ | $:-\ divides(M, N)$ | $\rightarrow filter(M, L)$ |
| $filter(M, N \cdot L)$ | $:-\ \mathbf{not}(divides(M, N))$ | $\rightarrow N \cdot filter'(M, L)$ |
| $divides(M, N)$ | $:-\ div(N, M, Q, 0)$ | |
| $filter(M, integers'(K))$ | | $\rightarrow filter(M, integers(K))$ |
| $sift(filter'(N, L))$ | | $\rightarrow sift(filter(N, L))$ |
| $filter(M, filter'(N, L))$ | | $\rightarrow filter(M, filter(N, L))$ |
| $prefix(Z, sift'(L))$ | | $\rightarrow prefix(Z, sift(L))$ |

In this way, simplification will always terminate for correct programs; any possible non-termination has been confined to the narrowing process.

## 5. CORRECTNESS

In this section we look at what it means for rewrite programs to be correct. Some related issues are considered in Zhang and Rémy (1985), Bergstra and Klop (1986), and Kaplan (1987).

### 5.1. Termination

As mentioned earlier, a correct rewrite system should not allow an infinite sequence of simplifications. Formally, we require that for some well-founded ordering $>$ on terms and for each rule $l :- p \rightarrow r$ we have $u[l\sigma] > p\sigma$ and $p\sigma$ implies $u[l\sigma] > u[r\sigma]$ for all contexts $u$ and substitutions $\sigma$ for variables in $l$. In particular, if a partial ordering $>$ on terms satisfies the conditions $u[t] > t$ and $s > t$ implies $u[s\sigma] > u[t\sigma]$ (for all terms $s$ and $t$, all contexts $u$, and all substitutions $\sigma$), and if for each rule $l :- p \rightarrow r$ we have $l > p, r$,

then termination is guaranteed. See Dershowitz (1987) for a survey of orderings used for proving termination of unconditional term-rewriting systems. An example of a non-terminating system is

$$a :- \neg(a = b) \rightarrow b;$$

it does not terminate, since the left-hand side $a$ also appears in the condition.

This termination requirement does not, however, preclude there being an infinite *narrowing* sequence. For example, the rules

$$listp(\mathbf{nil})$$
$$listp(A \cdot Y) :- listp(Y),$$

though terminating whenever used to simplify, go on generating solutions for the goal $listp(Z)$ forever.

For programs that mix simplification and narrowing by allowing variables in the condition $p$ that do not appear in the left-hand side $l$, we need to modify the termination requirements so that only substitutions that can result from narrowing are considered. Suppose the condition $p$ contains variables $\bar{X}$ that do appear in $l$ and variables $\bar{Y}$ that do not. Then we require, for some well-founded ordering $>$, that

$$u[l\sigma] > p\sigma$$
$$p\sigma\tau \quad \textit{implies} \quad u[l\sigma] > p\sigma\tau$$
$$p\sigma\tau \quad \textit{implies} \quad u[l\sigma] > u[r\sigma\tau],$$

for all rules $l :- p \rightarrow r$, contexts $u$, and substitutions $\sigma$ for $\bar{X}$ and $\tau$ for $\bar{Y}$. The first requirement guarantees that simplifying the condition will terminate; the second, that simplifying a solution to the condition terminates; the third, that for any solution, rewriting results in a smaller term. These conditions may be weakened further by considering only irreducible $\tau$.

### 5.2. Confluence

A rewrite system is said to be *confluent* if, whenever a term $t$ reduces to two terms $u$ and $v$, both $u$ and $v$ reduce to the same term $s$. Note that if a system is terminating and confluent, then every term reduces to a *unique* irreducible term. Terminating LISP-like programs (with mutually exclusive conditions that do not introduce free variables) are confluent, as are PROLOG-like programs (with only the term **true** for right-hand sides). An example of a non-confluent system is

$$X + 0 \rightarrow X$$
$$0 + s(Y) \rightarrow s(Y)$$
$$X + s(Y) \rightarrow s(X + Y).$$

The term $0 + s(Y)$ reduces to both $s(Y)$ and $s(0 + Y)$, neither of which are reducible.

For the purpose of computation, a system need not be confluent. As we will see shortly it is usually enough if it is *ground confluent,* i.e. if for every ground (variable-free) term $t$ (constructed from a given set of function symbols and individual constants) that reduces to two terms $u$

and $v$, both $u$ and $v$ reduce to the same term $s$. If a system is ground confluent and terminates for all (ground) terms, then every ground term reduces to a *unique* irreducible term. (We assume throughout that the set of ground terms is non-empty.) The above system is ground confluent, since all terms constructed from constant symbols 0, unary function symbols $s$, and at least one binary symbol $+$ are reducible, while those of the form $s(s(\ldots s(0)\ldots))$, containing no $+$, are irreducible. Interpreting 0 as zero, $s$ as successor, and $+$ as addition, each rule preserves equality. Hence, a term can reduce to only one of those irreducible terms. For 'syntactic' methods of demonstrating confluence of conditional systems, see Dershowitz *et al.* (1987) and Kaplan (1988); for 'semantic' methods, see Plaisted (1985).

## 5.3. Completeness

An interpreter is said to be *complete* for a logic-programming language, if for every logically satisfiable goal there is a successful computation path. The question we need to address now is under what conditions is conditional narrowing complete.

The answer to this question will be given in parts, for which we need to define a number of possible relations between terms for any given conditional rewrite system $R$.

(a) When we write $R \vdash (s = t)$, we mean that the equation $s = t$ follows *logically* (i.e. by first-order predicate calculus with equality) from the rules in $R$ interpreted as conditional equations. By $R \vdash p$, we mean that the formula $p$ (i.e. $p = \textbf{true}$) is provable from the rules in $R$. By the completeness of first-order predicate calculus with equality, provability and validity coincide.

(b) When we write

$$s \underset{c}{\overset{\supset}{\leftrightarrow}} t$$

we mean that, assuming $c$, the equation $s = t$ follows *conditionally* from the rules in $R$. That is, there are $n$ $(n \geq 0)$ instances $l_i' :- p_i' \rightarrow r_i'$ of rules in $R$ such that $R \vdash (c \supset p_i')$ for all $i$ $(1 \leq i \leq n)$ and $t$ can be obtained from $s$ by replacing occurrences of $l_i'$ with $r_i'$ or vice versa. By $s \overset{\supset}{\leftrightarrow} t$ we mean $s \underset{\textbf{true}}{\overset{\supset}{\longleftrightarrow}} t$.

(c) When we write

$$s \overset{\vee}{\leftrightarrow} t$$

we mean that the equation $s = t$ follows *by cases* from the rules in $R$. That is, there are $n$ $(n \geq 1)$ formulae $c_1, \ldots, c_n$ such that $R \vdash (c_1 \vee \cdots \vee c_n)$ and $s \underset{c_i}{\overset{\supset}{\leftrightarrow}} t$ for all $i$ $(1 \leq i \leq n)$.

(d) When we write

$$s \overset{*}{\leftrightarrow} t$$

we mean that the equation $s = t$ follows *equationally* from the rules in $R$. That is, there are $n$ $(n \geq 0)$ instances $l_i' :- p_i' \rightarrow r_i'$ of rules in $R$ such that $p_i' \overset{*}{\leftrightarrow}$ **true** for all $i$ $(1 \leq i \leq n)$ and $t$ can be obtained from $s$ by replacing occurrences of $l_i'$ with $r_i'$ or vice versa (and $\overset{*}{\leftrightarrow}$ is the least such relation).

(e) When we write

$$s \xrightarrow[c]{\supset} t$$

we mean that, assuming $c$, the term $s$ reduces to $t$ using the rules in $R$. That is, there are $n$ $(n \geq 0)$ instances $l_i' :- p_i' \rightarrow r_i'$ of rules in $R$ such that $R \vdash (c \supset p_i')$ for all $i$ $(1 \leq i \leq n)$ and $t$ can be obtained from $s$ by replacing occurrences of $l_i'$ with $r_i'$. By $s \xrightarrow{\supset} t$ we mean $s \xrightarrow[\textbf{true}]{\supset} t$.

(f) When we write

$$s \xrightarrow{*} t$$

we mean that $s$ *reduces* to $t$ using the rules in $R$. That is, there are $n$ $(n \geq 0)$ instances $l_i' :- p_i' \rightarrow r_i'$ of rules in $R$ such that $p_i' \xrightarrow{*}$ **true** for all $i$ $(1 \leq i \leq n)$ and $t$ can be obtained from $s$ by replacing occurrences of $l_i'$ with $r_i'$ (and $\xrightarrow{*}$ is the least such relation). The condition $p_i$ may contain variables not in $l_i$; we only insist that the particular instance $p_i'$ reduce to **true**. Note that for any two (ground) terms $s$ and $t$, if $s \overset{*}{\leftrightarrow} t$ and $R$ is (ground) confluent, then there is a term $u$ such that $s \xrightarrow{*} u$ and $t \xrightarrow{*} u$. If $t$ is irreducible, then it must be that $s \xrightarrow{*} t$.

(g) When we write

$$s \underset{\sigma}{\rightsquigarrow} t$$

we mean that $s$ *narrows* (in a single step) to $t$ via $\sigma$. That is, $s$ contains a non-variable subterm $u$ that unifies with the left-hand side $l$ of a rule $l :- p \rightarrow r$ in $R$ via a most general unifier $\mu$ (the variables of $l$ and $s$ are considered disjoint), $p\mu \overset{*}{\underset{\rho}{\rightsquigarrow}}$ **true**, and $\sigma$ is the composition of the two substitutions $\mu$ and $\rho$. By $s \overset{*}{\underset{\rho_1 \cdots \rho_n}{\rightsquigarrow}} t$ we mean that $s$ narrows to $t$ via $\rho_1 \cdots \rho_n$ in $n \geq 0$ steps, i.e.

$$s \underset{\rho_1}{\rightsquigarrow} \cdots \underset{\rho_n}{\rightsquigarrow} t.$$

In the unconditional case (see Dershowitz, 1985; Rėty *et al.*, 1985) one can show that narrowing with simplification is complete for any terminating ground confluent system (provided the term **true** is irreducible). For conditional systems the analogous result is that (under the same assumptions) any equationally satisfiable goal will be solved by conditional narrowing with simplification.

Firstly, with no assumptions regarding $R$, we can show that whenever $s\sigma$ reduces to $t$ for some irreducible substitution $\sigma$ and terms $s$ and $t$, it is the case that $s$ narrows to a term $u$ via some substitution $\nu$ such that $t$ is an instance of $u$ and $v$ is at least as general as $\sigma$. More precisely, if $s\sigma \xrightarrow{*} t$ and the terms that $\sigma$ substitutes for the variables in $s$ are irreducible, as are the substitutions $\sigma$ makes for any new variables introduced by conditions in the reduction sequence (as pointed out in Giovanetti and Moiso, 1987), then $s \overset{*}{\underset{v}{\rightsquigarrow}} u$, $u\tau = t$, and $v\tau = \sigma$, for some term $u$ and substitutions $v$ and $\tau$. This can be shown by induction on the *size* of the derivation $s\sigma \xrightarrow{*} t$ (that is, the number of rule applications in conditions as well as in terms): trivially, if the derivation is empty, then $s\sigma = t$, and an empty narrowing via an empty (identity) substitution gives the desired result. (Note that if $s$ is just a variable, then the derivation must be empty, since $\sigma$ is irreducible.) Suppose then that $s\sigma$ is first reduced using some rule $l :- p \rightarrow r$ in $R$. Since $\sigma$ is irreducible, it must be that $s$ has a non-variable subterm $u$ such that $u\sigma$ is an instance $l\theta$ of $l$, i.e. $s\sigma[u\sigma] = s\sigma[l\theta] \rightarrow s\sigma[r\theta] \xrightarrow{*} t$ and $p\theta \xrightarrow{*}$ **true** for some substitution $\theta$ of terms for the variables in $l$ and $p$. Let $\mu$ be the most general unifier of $u$ and $l$. Then, for some substitution $\tau$, $\sigma$ is $\mu\tau$ and $\theta$ is $\mu\tau$ when $\mu$ is restricted to variables in $l$. Note that $\tau$ is irreducible because $\sigma$ is. Since $p\theta = p\mu\tau \xrightarrow{*}$ **true**, by induction $p\mu \overset{*}{\underset{\rho}{\rightsquigarrow}}$ **true** and $\tau = \rho v$ for some irreducible substitution $v$. (If $p\mu$ narrows to some Boolean variable $X$, then $\rho$ needs to be composed with the substitution of **true** for $X$.) By the definition of a single narrowing step, $s[u] \underset{\mu\rho}{\rightsquigarrow} s\mu\rho[r\mu\rho]$. Since $s\sigma[r\theta] = s\mu\rho v[r\mu\rho v] \xrightarrow{*} t$, by induction $s\mu\rho[r\mu\rho] \overset{*}{\underset{\phi}{\rightsquigarrow}} u$, $v = \phi\psi$, and $t = u\psi$ for some substitution $\psi$. Together we have $s[u] \overset{*}{\underset{\mu\rho\phi}{\rightsquigarrow}} u$, $\sigma = \mu\rho\phi\psi$, and $t = u\psi$, as desired.

If $R$ is terminating and ground confluent, then whenever there is any (not necessarily irreducible) ground substitution $\sigma$ such that $s\sigma \xrightarrow{*} t$ and $t$ is irreducible, then there is also an irreducible substitution $\mu$ such that $s \overset{*}{\underset{\mu}{\rightsquigarrow}} u$ where $t$ is an instance of $u$ and $\mu$ is at least as general as a substitution $\sigma'$ to which $\sigma$ reduces. The following system is not ground confluent:

$$
\begin{array}{l}
r(a) \\
q(b) \\
\quad a \qquad \rightarrow b \\
p(X) :- q(X) \rightarrow r(X).
\end{array}
$$

Note that despite the fact that $p(a) \Rightarrow r(a) \Rightarrow$ **true**, since $q(a) \Rightarrow q(b)$. $p(X)$ narrows only to $r(b)$ and not beyond.

The straightforward simplification relation $\Rightarrow \cdots \Rightarrow$ is a special case of $\xrightarrow{*}$ in which new variables introduced by conditions are not substituted for (since that would require narrowing). A sequence of conditional narrowing steps from one unconditional goal to another corresponds to a single $\rightsquigarrow$ step, except that our operational semantics insists on fully simplifying terms and conditions before conditionally narrowing any further. For ground confluent $R$, this does not matter, since it can easily be shown that if $s \xrightarrow{*} u$ and $s \underset{\sigma}{\overset{*}{\rightsquigarrow}} t$, where $t$ is an irreducible ground term, and $\sigma'$ is a ground instance of $\sigma$, then $u \underset{\mu}{\overset{*}{\rightsquigarrow}} v$ for some substitution $\mu$ and term $v$ such that $t$ is an instance of $v$ and $\mu$ is at least as general as $\sigma'$. For programming purposes, $t$ is always the irreducible ground term **true**.

It follows from the above argument that for terminating and ground confluent $R$ and given goal $g$, if $g\sigma$ can be shown equationally to be equal to **true**, for some ground substitution $\sigma$, then from the goal rule

$$g \rightarrow answer(\bar{s})$$

conditional narrowing will generate an answer

$$\textbf{true} \rightarrow answer(\bar{s}\mu),$$

where $\mu$ is a substitution that is at least as general as one to which $\sigma$ reduces. For a similar result, see Kaplan (1988).

Next, we show that if conditions are restricted so that equations appear only as un-negated conjuncts, then whenever an equation $s = t$ follows logically from a satisfiable system $R$, it also follows by cases. This restriction is not adhered to by the system

$$c :- a = b \rightarrow a$$

$$c :- \neg(a = b) \rightarrow b$$

which has a negated equality in a conditional part. Though the equation $b = c$ is a logical consequence of the two conditional equations, it is not the case that $b \overset{\vee}{\leftrightarrow} c$.

For Boolean equations $s = t$, this result does not hold. We will discuss Boolean equations later. To prove this for non-Boolean equations, we assume that the syntax does not allow for non-Boolean operators to take Boolean operands and that standard Boolean simplifications for eliminating the *truth constants* **true** and **false** from Boolean terms (a small part of the propositional calculus system) are provided. Also, we assume that in an equation $u = v$, $u$ is Boolean if and only if $v$ is. (This proof does not, however, require that $R$ be confluent or terminating.)

Suppose that $R \vdash s = t$ for non-Boolean $s$ and $t$. Consider any variables in $s$ and $t$ to be Skolem constants. By Herbrand's theorem, for some finite conjunction $H$ of ground instances $e_i \wedge p_i \supset l_i = r_i$ of conditional equations in $R$, the formula $H \supset s = t$ is valid. Here $e_i$ is a conjunction of equations and $p_i$ is a Boolean expression. Consider the set of all ground predicates appearing in the $p_i$, $l_i$, and $r_i$. Let $D$ be a conjunction of each such predicate, or its negation, such that $R \wedge D$ is satisfiable. Note that $R$ logically implies the disjunction of all such $D$. Also, since $R$ is satisfiable, there is at least one such $D$. For any such $D$, the formula $(H \wedge D) \supset s = t$ must also be valid. Let $H'$ be the finite conjunction of conditional equations $e_i \supset l'_i = r'_i$ for which $D$ implies $p_i$, where $l'_i$ and $r'_i$ are $l_i$ and $r_i$, respectively, unless the latter are Boolean, in which case $l'_i$ ($r'_i$) is **true** if $D \supset l_i$ ($D \supset r_i$), and **false** otherwise. We have that $H' \wedge D$ is equivalent to $H \wedge D$ and, hence, that $(H' \wedge D) \supset s = t$. However, since non-Boolean operators may not have Boolean arguments, $D$ (which is satisfiable) does not influence the rest of this formula, so $H' \supset s = t$. Also, $H'$ is satisfiable, since $H \wedge D$ must be. Let $C$ be the conjunction of equations $l'_i = r'_i$ such that $H'$ implies $e_i$ and $D$ implies $p_i$; let $C'$ be the conjunction of (disjunctions of) inequations $\neg(e_i)$ such that $H'$ does not imply $e_i$ and $D$ does imply $p_i$. (In $C$, if $l'_i$ and $r'_i$ are Boolean then $l'_i = r'_i$ is interpreted as $l'_i \equiv r'_i$.) Now, $H'$ is a Horn set, since equations do not appear negated in conditions. Thus $H'$ (together with the equality axioms) has a minimal model, in which $C \wedge C'$ is true. In addition, $C \wedge C'$ implies $H'$, so $(C \wedge C') \supset s = t$. But $C'$ contains only inequations and so does not contribute to this formula (since $C \wedge C'$ is satisfiable). Thus, $C \supset s = t$. Also, if $l'_i$ and $r'_i$ are Boolean then the equation $l'_i = r'_i$ cannot contribute to the proof of $s = t$, since non-Boolean operators may not have Boolean arguments. Therefore, by the completeness of equational reasoning, $s = t$ is provable from non-Boolean equations in $C$ by replacement of equals by equals. Let $E$ be the conjunction of formulae $e_i \wedge p_i$ such that $H'$ implies $e_i$ and $D$ implies $p_i$. Then $D \wedge H$ implies $E$ (since $D \wedge H$ implies $H'$). Also, $s \underset{E}{\overset{\rightharpoonup}{\leftrightarrow}} t$ since $E$ implies the condition for each equation of $C$. Now, $H$ implies the disjunction of all such $D$, hence of all such $E$. Therefore, $s \overset{\vee}{\leftrightarrow} t$.

It remains to determine under what conditions any logically satisfiable goal is also equationally satisfiable. The crux of the problem is the inability to express equationally the fact that Boolean terms only take on the values **true** and **false**. We say that a theory is *disjunctively complete* for a rewrite system $R$ if for all true disjunctions $c_1 \vee \cdots \vee c_n$ of ground instances of conditions appearing in rules, at least one of the $c_i$ is itself true, i.e. if $R \vdash (c_1 \vee \cdots \vee c_n)$, then $R \vdash c_i$ for some $i$ $(1 \leq i \leq n)$. In particular, this condition holds if the theory has an initial model and all conditions are non-negated literals. It also holds if all ground instances of conditions are provably true or provably false. If $R$ is disjunctively complete, then whenever a ground equation $s = t$ follows by cases from $R$, it also follows conditionally. An example of a disjunctively incomplete system is

$$a :- p \rightarrow b$$

$$a :- \neg p \rightarrow b.$$

Though the equation $a = b$ is a logical consequence of the two conditional equations, it cannot be derived by replacement of equals for equals.

To see that $s \overset{\vee}{\leftrightarrow} t$ implies $s \overset{\rightharpoonup}{\leftrightarrow} t$ for disjunctively complete systems, consider the definitions. If $s \overset{\vee}{\leftrightarrow} t$, then for some $c_i$ such that $R \vdash \bigvee_i c_i$ we have

$s \underset{c_i}{\overset{\exists}{\leftrightarrow}} t$ for all $i$. Let $p'_{ij}$ be the instances of the conditions needed to show $s \underset{c_i}{\overset{\exists}{\leftrightarrow}} t$, in which case $R \vdash (c_i \supset \bigwedge_j p'_{ij})$. Hence, $R \vdash \bigvee_i \bigwedge_j p'_{ij}$, which can be re-expressed as $R \vdash \bigwedge_j \bigvee_i q'_{ji}$ for instances $q'_{ji}$ of the conditions. It follows that $R \vdash \bigvee_i q'_{ji}$ for each $j$ and, by disjunctive completeness, that $R \vdash q'_{jk}$, for some $k$. In other words, there are $k$ such that $R \vdash \bigwedge_j q'_{jk}$ and thus $s \overset{\exists}{\leftrightarrow} t$.

Note that if $R \vdash s = t$ implies $R \vdash s \overset{\vee}{\leftrightarrow} t$ and $R$ is disjunctively complete, then $R \vdash s = t$ implies $s \overset{\exists}{\leftrightarrow} t$, for ground terms $s$ and $t$. We would like to show that $R \vdash s = t$ implies $s \overset{*}{\leftrightarrow} t$. For this we consider a number of restrictions on $R$. We say that a system $R$ is *conditionally confluent* if for all ground terms $s$ and $t$, whenever $s \overset{\exists}{\leftrightarrow} t$, there exists a term $u$ such that $s \overset{\exists}{\rightarrow} u$ and $t \overset{\exists}{\rightarrow} u$. An *atom* is a non-equality predicate which is not a truth constant; a *literal* is an atom or its negation. We say that $R$ is *weakly complete for Booleans* if $R \vdash p$ implies $p \overset{\vee}{\leftrightarrow}$ **true** and $R \vdash \neg p$ implies $p \overset{\vee}{\leftrightarrow}$ **false**, and that it is *strongly complete for Booleans* if $R \vdash p$ implies $p \overset{*}{\leftrightarrow}$ **true** and $R \vdash \neg p$ implies $p \overset{*}{\leftrightarrow}$ **false**, both for all ground instances $p$ of atoms appearing in $R$. Note that both forms of Boolean completeness are true of Horn-clause programs and also if all atoms are reducible (as in Lisp-like programs) and $R$ is terminating.

Conditional confluence and weak Boolean completeness suffice for our overall completeness result. We say a Boolean equation $l = r$ is *simple* if $l$ is an atom and $r$ is a truth constant, and that a formula is *standard* if it is a conjunction of non-Boolean equations and Boolean literals. If for some terminating, conditionally confluent, and weakly Boolean-complete $R$ having standard conditions, it is the case that $R \vdash s = t$ implies $s \overset{\exists}{\leftrightarrow} t$ for all ground terms $s$ and $t$ such that $s$ is non-Boolean or $s = t$ is simple, then it must also be the case that $R \vdash s = t$ implies $s \overset{*}{\leftrightarrow} t$ for all such $s$ and $t$. This can be shown by induction. (The restriction that conditions be conjunction of equations and literals is not severe, since a rule $l :- c \vee d \rightarrow r$ can be re-expressed as two rules, $l :- c \rightarrow r$ and $l :- d \rightarrow r$. Also, a non-simple Boolean rule $p :- c \rightarrow e$ may be transformed into the rules $p :- c \wedge e \rightarrow$ **true** and $p :- c \wedge \neg e \rightarrow$ **false**, and then disjunctions in $e$ can be eliminated as above.) On the other hand, for the terminating non-confluent program

$$\begin{array}{l} q \\ q :- p \quad \rightarrow p \\ q :- \neg p \rightarrow p \end{array}$$

we have $p \overset{\exists}{\leftrightarrow}$ **true**, but not $p \overset{*}{\leftrightarrow}$ **true**. Similarly, for the non-terminating confluent system

$$p :- p$$

$$p :- \neg p$$

we have $p \overset{\supseteq}{\leftrightarrow}$ **true**, but not $p \overset{*}{\leftrightarrow}$ **true**.

Note that conditional confluence implies ground confluence for terminating systems $R$ such that $R \vdash s = t$ implies $s \overset{\supseteq}{\leftrightarrow} t$. As noted above, if $R$ is ground confluent, then $s \overset{*}{\leftrightarrow}$ **true** implies $s \overset{*}{\rightarrow}$ **true**, for any ground term $s$, since **true** is irreducible (it may not appear alone on the left-hand of any rule).

To recapitulate, we have shown that if $R$ is satisfiable, terminating, conditionally confluent, disjunctively complete, weakly complete for Booleans, and has standard conditions, then our interpreter is complete for any logically satisfiable standard goal. These conditions are satisfied by correct LISP-like and PROLOG-like programs.

We can also show that if $R \vdash s = t$ implies $s \overset{\supseteq}{\leftrightarrow} t$ for all ground terms $s$ and $t$, then $R \vdash s = t$ implies $s \overset{*}{\leftrightarrow} t$ for all such $s$ and $t$, provided that $R$ possesses the stronger form of Boolean completeness, is ground confluent (but not necessarily terminating), conditions are standard, and Boolean rules are simple. Without disjunctions in conditions, this means that equality reasoning suffices for establishing conditions, and by the completeness of positive-unit resolution for equational Horn clauses, the result follows. Under these conditions, for any non-Boolean ground equations $s = t$ which follows logically from a satisfiable and disjunctively complete $R$, there is some term $u$ such that $s \overset{*}{\rightarrow} u$ and $t \overset{*}{\rightarrow} u$. (If confluence, disjunctive completeness, and Boolean completeness hold for non-ground terms, then these results may be extended to non-ground $s$ and $t$.)

## 6. IMPLEMENTATION

Currently, we have an experimental implementation in FRANZ LISP (Foderaro *et al.*, 1984), named RITE. See Josephson and Dershowitz (1986) for details. In this section, we briefly touch on some of the engineering issues that are addressed.

### 6.1. Built-in functions

With full functional notation, one can make full use of built-in functions (cf. Futatsugi *et al.*, 1985). An example is the binary-search program.

Given a goal $bin(Z, x, a, n)$ and a non-decreasing function $f$, this program searches for a $Z$ among the positions $a, a+1, \ldots, a+n-1$, such that $f(Z) \leq X < f(Z+1)$. To use this program, a system for computing $f$ must also be provided; it presumes that $f(a) \leq x < f(a+n)$.

*Binary search*

$$\begin{aligned}
&bin(Z, X, Z, 1)\\
&bin(P, X, Z, Y) :- \mathbf{less}(X, f(mid(Z, Y)))\\
&\qquad \rightarrow bin(P, X, Z, half(Y))\\
&bin(P, X, Z, Y) :- \mathbf{even}(Y), \mathbf{not}(\mathbf{less}(X, f(mid(Z, Y))))\\
&\qquad \rightarrow bin(P, X, mid(Z, Y), half(Y))\\
&bin(P, X, Z, Y) :- \mathbf{odd}(Y), \mathbf{not}(\mathbf{less}(X, f(mid(Z, Y))))\\
&\qquad \rightarrow bin(P, X, mid(Z, Y), half(Y) + 1)\\
&mid(I, J) :- \mathbf{number}(I, J)\\
&\qquad \rightarrow \mathbf{plus}(I, half(J))\\
&half(J) :- \mathbf{number}(J)\\
&\qquad \rightarrow \mathbf{quotient}(J, 2)\\
&I + 1 :- \mathbf{number}(I)\\
&\qquad \rightarrow \mathbf{add1}(I)
\end{aligned}$$

Built-in functions are not narrowable, i.e. they only work when given constructor arguments. Note that one can mix the use of built-in functions (when the arguments are fully evaluated) and defined functions (that can be applied even to nonground terms) as in:

$$\begin{aligned}
&I + 0 && \rightarrow I\\
&I + J :- \mathbf{number}(I, J) && \rightarrow \mathbf{plus}(I, J)\\
&I + J = K :- \mathbf{number}(I, K) && \rightarrow J = \mathbf{diff}(K, I).
\end{aligned}$$

**6.2. Destructive assignment**

Since there is no backtracking over simplifications, arrays can be handled cheaply by destructive assignments. Consider the *array increment* program.

*Array increment*

$$\begin{aligned}
&incr(A, I, N) :- \mathbf{less}(I, N) \rightarrow incr(\mathbf{assign}(A, I, A[I] + 1), \mathbf{add1}(I), N)\\
&incr(A, N, N) \qquad\qquad\quad \rightarrow \mathbf{assign}(A, N, A[N] + 1)
\end{aligned}$$

In applying it to a goal containing the subterm

$incr(array, 0, 9)$,

there is no need to preserve old values of *array*. If, say, *array*[*i*] = *i*, then

$$\begin{aligned}&incr(array, 0, 9) \Rightarrow \cdots \Rightarrow incr(\mathbf{assign}(array, 0, 1), 1, 9)\\ &\qquad \Rightarrow \cdots \Rightarrow incr(\mathbf{assign}(\mathbf{assign}(array, 0, 1), 1, 2), 2, 9) \Rightarrow \cdots.\end{aligned}$$

As long as the array $A$ only appears within the built-in array assignment function, its intermediate values may be overwritten.

## 6.3. Queuing

The effect of a single rule application on future applications is localized: the only *new* possibilities for applications are within the new subterm introduced by the right-hand side of a rule and at nearby, enclosing function symbols. Thus, in most cases only an area bounded by the size of the rules in the program needs to be examined at each step. This suggests maintaining a queue of positions at which rules can be applied, rather than searching through the whole term again and again. Congruence-closure algorithms (Nelson and Oppen, 1980), which work on ground terms, go a step further and speed things up by remembering partial matches, obviating the need to re-examine enclosing function symbols. Our implementation (Josephson and Dershowitz, 1986) avoids searching through terms by preprocessing the program and setting up demons that let partial matches progress.

There is a space–time tradeoff in deciding whether an implementation should save the results of previous simplifications. OBJ2 (Futatsugi *et al.*, 1985), for example, maintains a hash-table of terms and their fully simplified forms.

## 6.4. Parallelism

Confluence guarantees that the order in which simplifications are applied is immaterial, making simplification of non-overlapping subterms a natural candidate for concurrent execution. No communication between processes or storage overhead would be required. Alternative narrowings, on the other hand, can lead to success or failure; to guarantee that an existing answer will be found (see Section 5.3) requires that no possible narrowing be 'starved'. Ensuring this by breadth-first search, however, would, in general, make heavy storage demands.

There are cases when certain potential narrowings are sure to be redundant and can be eliminated. Particularly with parallel execution, it would be desirable to prune such unproductive paths. By including rules for false cases, as outlined above, *unsatisfiable* goals will not be pursued (cf. Fribourg, 1985). By not just narrowing goals, but also comparing one with another, duplicate goals can be pruned. In particular, given two rules for some subgoal $g$, one of the unconditional form

$$g \rightarrow answer(\bar{s}),$$

and the other of the more restrictive form

$$g\sigma :- p \rightarrow answer(\bar{t}),$$

where $\sigma$ is any substitution, the latter rule can be ignored—assuming any one solution suffices (cf. Réty *et al.*, 1985). Similarly, using program rules to overlap assertions, as well as goals, provides a 'forward reasoning' capability, generating new facts from old ones (cf. Dershowitz, 1985).

## 7. DISCUSSION

The approach outlined here is an attempt to combine features of functional and logic programming in a unified way. Functional notation and an evaluation mechanism are borrowed from functional programming; free variables and narrowing are added so that solutions to a given goal can be sought. Pattern-directed invocation and assignment by unification are borrowed from logic programming; unification within literals and replacement of subterms are added so that equations can be handled.

For a programming language to be justifiably called 'logical', it ought to aspire to have a declarative and sound semantics, and a complete and efficient interpreter. That is, each statement should have a local declarative meaning and each procedural step should follow logically from the meaning of the statements. Any logically satisfiable goal should be solvable by an interpreter with predictable (and reasonable) time and space requirements for executing a single statement. We believe that the language proposed in this paper comes reasonably close to meeting these criteria.

Various proposals have been made over the past few years for combining features of functional programming and logic (relational) programming. Two surveys are Bellia and Levi (1986) and Reddy (1986). See also the collection in DeGroot and Lindstrom (1986).

In some cases, the proposed language provides a convenient interface between resolution-based goal reduction and rewrite-based term evaluation, without fully integrating the two. A few examples are QLOG (Komorowski, 1982), LOGLISP (Robinson and Sibert, 1982), the original Qute (Sato and Sakurai, 1983), and Funlog (Subrahmanyam and You, 1984). In these languages, terms are rewritten to normal forms before unification is attempted. Function definitions do not, however, cause free variables to be instantiated during goal reduction. Consequently, such languages are incomplete, in the sense that a solution to a goal will not necessarily be found whenever one provably exists. Prolog-with-Equality (Kornfeld, 1983) allows the programmer to explicitly include additional facts about functions—to be used in solving goals—but is not inherently complete.

Paramodulation (unifying one side of an equation with a non-variable subterm of a clause and replacing with the other side) is the natural and customary way of handling equality in resolution-based theorem provers, and—for some proof strategies, at least—is a (refutationally) complete theorem-proving method. Uniform (Kahn, 1981) is an early combination of LISP and PROLOG, incorporating such an equality rule. Paramodulation can be simulated by resolution when functions are represented by predicates and terms are decomposed (Brand, 1975; see also Plaisted and Greenbaum, 1984); an approach like this to solving goals with equalities is taken in Cox and Pietrzykowski (1985).

Though completeness is achievable (in the first-order case, at least) by harnessing a full-fledged theorem prover to the search for solutions, that would not constitute a *programming* language. In our opinion, an interpreter for a language—as contrasted with a theorem prover—should not be required to deduce new facts from comparisons between program statements, nor draw conclusions from alternative subgoals generated by a non-deterministic computation. (A compiler, on the other hand, may do some such things for reasons of efficiency.) In the purely relational case (Kowalski, 1974), the need for any kind of forward reasoning is avoided by insisting that program statements be Horn clauses; then, a 'linear' resolution strategy suffices for completeness. A similar sort of linear operational semantics is desirable in the presence of function definitions. The proposals for handling functions and equality in Dershowitz (1982), Durand (1984), and Fribourg (1984) require non-linear reasoning (see also Bandes, 1984).

Whereas paramodulation uses both sides of an equation in the same way, narrowing is more directed, allowing unification just with left-hand sides. Narrowing, too, can be simulated (in PROLOG) by decomposing terms, as done by Deransart (1983). A programming language with narrowing-like operational semantics was first suggested by Dershowitz (1983, 1984); its implementation is described in Dershowitz and Josephson (1984). There, both Horn clauses and function definitions were expressed as unconditional rules and 'linear completion' (a slight generalization of narrowing which helps handle conditions) was used to solve goals. Independent suggestions of narrowing, or close variants thereof, for languages incorporating functions and goal reduction include: the use of 'clausal superposition' for Horn clauses with equality in SEC (Fribourg, 1984); the interleaving in Eqlog (Goguen and Meseguer, 1984, 1986) of narrowing for solving equations with linear resolution for Horn clauses; the use of an 'equality rule' within TABLOG (Malachi *et al.*, 1984); unification with conditional expressions in (the newer) Qute (Sato and Sakurai, 1984); the use of oriented, decomposed equations within PROLOG by Tamaki (1984); the use of 'logical variables' in the deterministic functional language FGL + LV (Lindstrom, 1985); and the constructor-based 'object refinement' method used in EqL (Jayaraman and Silbermann, 1986). Some related proposals are Dershowitz and Plaisted (1985), Kanamori (1985), McCabe (1985), Reddy (1985), Smolka (1985), Barbuti *et al.* (1986), Darlington *et al.* (1986), You and Subrahmanyam (1986), and Robinson (1987).

Using conditional (if-then-else) terms, one can always express conditional equations (or equivalences) unconditionally. For completeness of narrowing, (ground) confluence of the system of oriented equations is required. With confluence, any irreducible solution to a goal can be found by narrowing. (Without confluence, one could only say that narrowing must come up with a solution if there is an irreducible substitution which, applied to the goal, gives an instance that simplifies to 'true'. See Section 5.3.) Happily, there are syntactic means of ensuring (full) confluence for unconditional systems of rules, by restricting overlaps of left-hand sides and allowing, on the left, only one occurrence of each variable. Pragmatically, these relatively severe restrictions also make it easier to determine which rules are applicable at each stage of the computation. Proposals imposing such restrictions include Reddy (1985), Jayaraman and Silbermann (1986), and You and Subrahmanyam (1986).

The problem with using conditional terms to encode conditional equations is that the resultant unconditional equations are not, in general, terminating rewrite rules—even when the corresponding conditional rules would be. Without termination, solutions having no irreducible normal form may be lost (but see You and Subrahmanyam, 1986). More importantly, it cannot be said that rewriting 'simplifies' anything, and uninhibited rewriting need not lead to an irreducible term. (Rewriting is the special case of narrowing in which no free variables appearing in the goal are instantiated.) Thus, alternative rewrite paths must be explored, even if one is satisfied just with irreducible solutions. (Actually, given confluence, rewriting of goals need never be undone, and it suffices to try all narrowing paths after any *finite* amount of rewriting.)

Using conditional equations, on the other hand, allows one to program with rules that can never lead to infinite sequences of rewrites. For conditional rules, appropriate notions of rewriting, narrowing, termination, and confluence have been developed (see Section 5). With termination and confluence, full advantage can be taken of the evaluation mechanism available to functional languages, restricting narrowing to irreducible terms, without jeopardizing completeness.

We contend that predicating completeness on the programmer's defining only terminating functions is justified, since any potentially infinite computations may be coded so as to be handled by the narrowing mechanism (as we illustrated in Section 4.5). Instead of demanding full confluence, we prefer to make the guarantee of completeness subject to the programmer's supplying a ground confluent system of rules. Ground confluence is essentially a consistency requirement; it means that different ways of evaluating the same ground term can not result in distinct values (ground normal forms). Methods for establishing ground confluence are given by Plaisted (1985), Zhang and Rémy (1985), and others. Syntactic methods for full confluence are also available (Dershowitz *et al.*, 1987; Kaplan, 1988).

Distinguishing between narrowing (by unification) and rewriting (by pattern matching), provides us with a clean, run-time distinction between 'don't know' and 'don't care' non-determinism. Rewriting may proceed in any fashion and no immediate results need be remembered (the *don't care* part); narrowing is used to explore alternative routes when the right path to a solution is unknown (the *don't know* part) and need only be resorted to when further rewriting is no longer possible. In this way, unneeded backtracking is avoided and many narrowing paths share the results of one rewrite path (see Josephson and Dershowitz, 1986).

Simplification, that is rewriting via terminating rules, is a very powerful feature, particularly when—as in our language—defined function symbols are allowed to be arbitrarily nested in left-hand sides. It is, of course, important not to incur heavy costs in searching for applicable rewrites. We believe that it is possible to minimize the overhead involved in various ways, including taking advantage of the fact that rewrites that fail only because of a mismatch with a free variable signify a potential narrowing (see Josephson and Dershowitz, 1986). Exactly how much simplification is performed before each narrowing step is a matter of taste, since completeness is not affected by this decision. Narrowing only fully simplified goals has been advocated, in the unconditional case, by Dershowitz (1983) and Rέty *et al.* (1985), and in the conditional case, by Dershowitz and Plaisted (1985) and Fribourg (1985). In our current implementation, we do not solve (i.e. narrow) conditions containing new variables when looking for applicable rewrites; Fribourg (1984, 1985) had simplification by unconditional rules only; at most one rewrite step precedes narrowing in the implementation described by Fribourg (1986).

Questions of efficiency and control are paramount in the design of a practical programming language. The avoidance of 'occur checks' when attempting to unify terms, as in FGL+LV (Lindstrom, 1985), 'surface deduction' (Cox and Pietrzykowski, 1985), LEAF (Barbuti *et al.*, 1986), and EqL (Jayaraman and Silbermann, 1986), trades off logical soundness (in the traditional sense) for efficiency and can cause spurious (and unprintable) solutions to be found. TABLOG (Malachi *et al.*, 1984), on the other hand, employs true unification, with 'occur check'. We recommend avoiding the expensive check only when soundness is maintained; see Plaisted (1984).

Even if a program is confluent and terminating, alternative narrowing derivations must be explored if completeness is to be assured. Narrowing-based languages, such as FGL + LV (Lindstrom, 1985), which deterministically choose one possible narrowing over others, cannot guarantee that solutions will be found. Restrictions and variations of narrowing which do preserve completeness are described in Hullot (1980) and Martelli *et al.* (1986). Some superfluous narrowing paths can be avoided by making a distinction between constructor symbols and defined ones (assuming that terms built entirely from constructors are irreducible). Two terms headed by different constructors can never be equal; when headed by the same constructor, they are equal if, and only if, their respective arguments are equal. This distinction can be programmed in with appropriate 'eager' simplification rules as illustrated in Dershowitz and Plaisted (1985) and described in Fribourg (1985); generated automatically, as in Josephson and Dershowitz (1986); or built in—usually with a 'lazy' narrowing strategy—as in Reddy (1985), Kanamori (1985), Jayaraman and Silbermann (1986), and Barbuti *et al.* (1986).

To force conditions to be evaluated (or solved) before the branches of a conditional expression, some authors impose a leftmost strategy (e.g. Reddy, 1985; You and Subrahmanyam, 1986). Our approach is based, instead, on conditional rules, which give the programmer a measure of local control over the computation by letting a rewrite or narrowing go through only when the condition holds. SLOG (Fribourg, 1985) and Eqlog (Goguen and Meseguer, 1986) are similar in this regard. Deransart (1983) showed how to effect conditional narrowing within Prolog; Dershowitz and Plaisted (1985) and Fribourg (1985) combine conditional narrowing with eager rewriting.

With terminating rules, there is no need for an outermost (or lazy) evaluation strategy to ensure that a value for a term will be reached. Assuming ground confluence and termination, any strategy can be used for simplification. For narrowing, TABLOG (Malachi *et al.*, 1984), for example, adopted an innermost-first search strategy. We believe that avoiding a declared strategy will encourage more logical logic-programming, by preventing the programmer from presuming anything about the global context in which a rewrite or narrowing is performed. Instead, our language provides an order-independent notion of simplification, over and above narrowing. SLOG (Fribourg, 1985) also incorporates eager simplification, but always chooses the leftmost-innermost narrowing path, and hence is complete only in certain situations.

Some narrowing languages, e.g. Fresh (Smolka, 1985), implement negation 'by failure'. We prefer (as does Fribourg, 1985) to handle negation by incorporating negative information in the form of rewrite rules, which are then used to simplify subgoals to 'false'. Combined with eager simplification, this approach has the advantage of allowing unsatisfiable goals to be pruned, thereby avoiding some potentially infinite narrowing paths (and should allow for finite representation of solution sets in more cases than would be possible without simplification). An equational approach to negation is also taken in Goguen and Meseguer (1984). In the purely equational Horn-clause case, ground confluence and termination are necessary and sufficient for completeness of conditional narrowing (with or without simplification); in this paper, we have also investigated completeness in more general situations.

Narrowing-based languages with extensions for handling set constructs include: Goguen and Meseguer (1984), Darlington *et al.* (1986), and Jayaraman and Silbermann (1986). Two languages that incorporate 'higher-order' functions are Fresh (Smolka, 1985) and IDEAL (Bosco and Giovannetti, 1986). An 'order-sorted' logic underlies Eqlog (Goguen and Meseguer, 1984, 1986). Preprocessing and compilation techniques for rewriting and narrowing are explored in Fribourg (1987), Josephson and Dershowitz (1987), and Kaplan (1987).

**Acknowledgements**

We thank Alan Josephson for his suggestions and for his implementation, RITE, Uday Reddy and Jean-Luc Rémy for stimulating discussions, and Joe Goguen, Naomi Lindenstrauss, Eric Muller, and Jia You for their comments. This research was supported in part by the National Science Foundation under Grants DCR 85-13417 and DCR 85-16243.

## REFERENCES

Bandes, R. G. (1984) Constraining-unification and the programming language Unicorn.. *Proc. Eleventh ACM Symp. on Principles of Programming Languages,* Salt Lake City, Utah, pp. 106–10. Also (1986) in *Logic programming* (eds. D. DeGroot and G. Lindstrom) pp. 397–410. Prentice-Hall, Englewood Cliffs, N.J.

Barbuti, R., Bellia, M., Levi, G., and Martelli, M. (1986) LEAF: a language which integrates logic, equations and functions. In *Logic programming* (eds D. DeGroot and G. Lindstrom) pp. 201–38. Prentice-Hall, Englewood Cliffs, N.J.

Bellia, M. and Levi, G. (1986) The relation between logic and functional languages: a survey. *J. Logic Programming* 3(3), 217–36.

Bergstra, J. A. and Klop, J. W. (1986) Conditional rewrite rules: confluence and termination. *J. Computer Syst. Sci.* **32,** 323–62.

Bosco, P. G. and Giovannetti, E. (1986) IDEAL: an ideal deductive applicative language. *Proc. IEEE Symp. on Logic Programming,* Salt Lake City, Utah, pp. 89–94.

Brand, D. (1975) Proving theorems with the modification method. *SIAM J. Computing* **4,** 412–30.

Burstall, R. M., MacQueen, D. B., and Sannella, D. T. (1980) HOPE: an experimental applicative language. *Conference Record of the 1980 LISP Conference,* Stanford, Calif. pp. 136–43.

Clark, K. L. (1978) Negation as failure. In *Logic in data bases* (eds H. Gallaire and J. Minker) pp. 292–322. Plenum Press, New York.

Clocksin, W. F. and Mellish, C. S. (1984) *Programming in Prolog,* 2nd edn. Springer-Verlag, New York.

Darlington, J., Field, A. J., and Pull, H. (1986) The unification of functional and logic languages. In *Logic programming: functions, relations, and equations* (eds D. DeGroot and G. Lindstrom) pp. 37–70. Prentice-Hall, Englewood Cliffs, N.J.

DeGroot, D. and Lindstrom, G., eds. (1986) *Logic programming: functions, relations, and equations,* Prentice-Hall, Englewood Cliffs, N.J.

Dershowitz, N. (1982) Applications of the Knuth–Bendix completion procedure. *Proc. seminaire d'informatique theorique,* Paris, pp. 95–111.

Dershowitz, N. (1983) Computing with rewrite systems. *Proc. NSF Workshop on the Review Rule Laboratory,* Schenectady, N.Y., pp. 269–98. Revised version (1985) in *Information and Control* 64 (2/3), 122–57.

Dershowitz, N. (1984) Equations as programming language. *Proc. Fourth Jerusalem Conference on Information Technology,* Jerusalem, pp. 114–24.

Dershowitz, N. (1987) Termination of rewriting. *J. Symbolic Computation* 3(1/2), 69–115.

Dershowitz N. and Josephson, N. A. (1984) Logic programming by completion. *Proc. Second Int. Logic Programming Conf.,* Uppsala, pp. 313–20.

Dershowitz, N., Okada, M., and Sivakumar, G. (1987) Confluence of conditional rewrite systems. *Proc. First Int. Workshop on Conditional Term Rewriting Systems,* Orsay. In press.

Dershowitz, N. and Plaisted, D. A. (1985) Logic programming *cum* applicative programming. *Proc. IEEE Symp. on Logic Programming,* Boston, Mass., pp. 54–66.

Durand, J. (1984) Une stratégie de réécriture pour les programmes logiques. Rapport 84-R-029, Centre de Recherche en Informatique de Nancy, Nancy.

Fages, F. (1984). Associative–commutative unification. *Proc. Seventh Int. Conf. on Automated Deduction,* Napa, Calif., pp. 194–208.

Foderaro, J. K., Sklower, K. L., and Layer, K. (1984) The Franz Lisp manual. In *Unix programmer's manual: supplementary documents* (eds M. J. Karels and S. J. Leffler). University of California, Berkeley, Calif.

Fribourg, L. (1984) Oriented equational clauses as a programming language. *Proc. Eleventh EATCS Colloquium on Automata, Languages, and Programming,* Antwerp, pp. 162–73. Revised version in *J. Logic Programming* **1** (2) 179–210.

Fribourg, L. (1985) Slog: a logic programming language interpreter based on clausal superposition and rewriting. *Proc. 1985 Symp. on Logic Programming,* Boston, Mass., pp. 172–184. Earlier version in *Proc. First Int. Conf. on Rewriting Techniques and Applications,* Dijon, pp. 325–44.

Fribourg, L. (1986) Prolog with simplification. Report 86–41, LITP, Paris.

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P., and Meseguer, J. (1985) *Principles of OBJ2. Conference Record of the Twelfth ACM Symp. on Principles of Programming Languages,* New Orleans, La., pp. 52–66.

Giovannetti, E. and Moiso, C. (1987) A completeness result for conditional narrowing. *proc. First Int. Workshop on Conditional Term Rewriting Systems,* Orsay. In press.

Goguen, J. A. and Meseguer, J. (1984) Equality, types, modules and generics for logic programming. *Proc. Second Int. Logic Programming Conf.,* Uppsala, pp. 115–25. Revised version in *J. Logic Programming* 1(2), 179–210.

Goguen, J. A. and Meseguer, J. (1986) Eqlog: equality, types, and generic modules for logic programming. In *Logic programming* (eds D. DeGroot and G. Lindstrom) pp. 295–363. Prentice-Hall, Englewood Cliffs, N.J.

Gordon, M., Milner, R., and Wadsworth, C. (1979) Edinburgh LCF. *Lecture notes in computer science,* Vol. 78, Springer-Verlag, Berlin.

Hewitt, C. (1971) Description and theoretical analysis (using schemata) of Planner: a language for providing theorems and manipulating models in a robot. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Mass.

Hsiang, J. and Dershowitz, N. (1983) Rewrite methods for clausal and non-clausal theorem proving. *Proc. Tenth EARCS Int. Colloquium on Automata, Languages, and Programming,* Barcelona, pp. 331–46.

Huet, G. and Oppen, D. C. (1980) Equations and rewrite rules: a survey. In *Formal* *language theory: perspectives and open problems* (ed. R. Book) pp. 349–405. Academic Press, New York.

Hullot, J.-M. (1980) Canonical forms and unification. *Proc. Fifth Conf. on Automated Deduction,* Les Arcs, pp. 318–34.

Jayaraman, B. and Silbermann, F. S. K. (1986) Equations, sets, and reduction semantics for functional and logic programming. *Proc. ACM Conf. on LISP and Functional Programming,* Cambridge, Mass., pp. 320–31.

Josephson, N. A. and Dershowitz, N. (1986) An efficient implementation of narrowing: The RITE way. *Proc. IEEE Symp. on Logic Programming,* Salt Lake City, Utah, pp. 187–97. Revised version to appear in *J. Logic Programming.*

Kahn, K. N. (1981) Uniform—a language based upon unification which unifies much of Lisp, Prolog and Act 1. *Proc. Seventh Int. Joint Conf. on Artificial Intelligence,* Vancouver, B. C., pp. 933–9. Revised version (1986) in *Logic programming* (eds. D. DeGroot and G. Lindstrom) pp. 411–38. Prentice-Hall, Englewood Cliffs, N.J.

Kanamori, T. (1985) Computation by meta-unification with constructors. Report TR-152, Institute for New Generation Computer Technology, Tokyo.

Kaplan, S. (1987) Simplifying conditional term rewriting systems. *J. Symbolic Computation.* In press.

Kaplan, S. (1987) A compiler for conditional term rewriting systems. *Proc. Second Int. Conf. on Rewriting Techniques and Applications,* Bordeaux, pp. 25–41.

Kapur, D., Sivakumar, G., and Zhang, H. (1986) RRL: a rewrite rule laboratory. *Proc. Eighth Int. Conf. on Automated Deduction,* Oxford, pp. 691–692.

Knuth, D. E. and Bendix, P. B. (1970) Simple word problems in universal algebras. In *Computational problems in abstract algebra* (ed. J. Leech) pp. 263–97. Pergamon Press, Oxford.

Kornfeld, W. (1983) Equality for PROLOG. *Proc. Eighth Int. Joint Conf. on Artificial Intelligence,* Karlsruhe, pp. 514–9. Revised version (1986) in *Logic Programming* (eds D. DeGroot and G. Lindstrom) pp. 279–93. Prentice-Hall, Englewood Cliffs, N.J.

Kowalski, R. A. (1974) Predicate logic as programming language. *Proc. IFIP Congress,* Amsterdam, pp. 569–74.

Lindstrom, G. (1985) Functional programming and the logical variable. *Proc. Twelfth ACM Symp. on Principles of Programming Languages,* New Orleans, La., pp. 266–80.

Malachi, Y., Manna, Z., and Waldinger, R. J. (1984) TABLOG: the deductive tableau programming language. *Proc. ACM Lisp and Functional Programming Conf.* Austin, Texas, pp. 323–30. Revised version (1986) in *Logic Programming* (eds D. DeGroot and G. Lindstrom) pp. 365–94. Prentice-Hall, Englewood Cliffs, N.J.

McCabe, F. G. (1985) Lambda PROLOG. Report, Department of Computing, Imperial College, London.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. (1965) *LISP 1.5 programmer's manual.* MIT, Cambridge, Mass.

Nelson, C. G. and Oppen, D. C. (1980) Fast decision procedures based on congruence closure. *J. Association for Computing Machinery* **27**(2), 356–64.

O'Donnell, M. J. (1985) *Equational logic as a programming language.* MIT Press, Cambridge, Mass.

Plaisted, D. A. (1984) The occur-check problem in Prolog. *New Generation Computing* **2**(4), 309–322.

Plaisted, D. A. (1985) Semantic confluence tests and completion methods. *Information and Control* **65**(2/3), 182–215.

Plaisted, D. A. and Greenbaum, S. (1984) Problem representations for backchaining and equality in resolution theorem proving. *Proc. First Annual AI Applications Conference,* Denver, Colo., pp. 417–23.

Reddy, U. (1985) Narrowing as the operational semantics of functional languages. *Proc.*

*1985 Symp. on Logic Programming,* Boston, Mass., pp. 138–51.

Reddy, U. S. (1986) On the relationship between logic and functional languages. In *Logic programming* (eds D. DeGroot and G. Lindstrom) pp. 3–36. Prentice-Hall, Englewood Cliffs, N.J.

Réty, P., Kirchner, C., Kirchner, H., and Lescanne, P. (1985) NARROWER: a new algorithm for unification and its application to logic programming, *Proc. First Int. Conf. on Rewriting Techniques and Applications,* Dijon, pp. 141–57.

Robinson, J. A. (1963) Theorem proving on the computer. *J. Association for Computing Machinery* **10**(2), 163–74.

Robinson, J. A. (1987) Beyond LOGLISP: combining functional programming. In *Machine intelligence 11* (eds J. E. Hayes, D. Michie, and J. Richards) pp. 57–67. This volume.

Sato, M. and Sakurai, T. (1984) QUTE: a functional language based on unification. *Proc. Int. Conf. on Fifth Generation Computer Systems,* Tokyo, pp. 157–65. Revised version (1986) in *Logic programming* (eds D. DeGroot and G. Lindstrom) pp. 131–55. Prentice-Hall, Englewood Cliffs, N.J.

Shapiro, E. Y. (1983) A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo.

Slagle, J. R. (1974) Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. Association for Computing Machinery* **21**(4), 622–42.

Smolka, G. (1985) FRESH: a higher-order language with unification and multiple results. In *Logic programming: functions, relations and equations* (eds D. DeGroot and G. Lindstrom) pp. 469–524. Prentice-Hall, Englewood Cliffs, N.J.

Stickel, M. E. (1981) A unification algorithm for associative-commutative functions. *J. Association for Computing Machinery* **28**(3), 423–34.

Subrahmanyam, P. A. and You, J.-H. (1984) FUNLOG = functions + logic: a computational model integrating functional and logic programming. *Proc. IEEE Symp. on Logic Programming,* Atlantic City, N.J., pp. 144–53. Revised version (1986) in *Logic Programming* (eds D. de Groot and G. Lindstrom), pp. 157–98. Prentice-Hall, Englewood Cliffs, N.J.

Tamaki, H. (1984) Semantics of a logic programming language with a reducibility predicate. *Proc. IEEE Symp. on Logic Programming,* Atlantic City, N.J., pp. 259–64.

Turner, D. A. (1979) SASL language manual. Report CS/79/3, Department of Computational Science, University of St. Andrews, St. Andrews.

Zhang, H. and Rémy, J.-L. (1985) Contextual rewriting. *Proc. First Int. Conf. on Rewriting Techniques and Applications,* Dijon, pp. 46–62.

3

# Beyond LOGLISP: combining functional and relational programming in a reduction setting

J. A. Robinson
School of Computer and Information Science, Syracuse University, USA

The initial plan for LOGLISP [1] was simply that it would offer, within LISP, a Horn-clause relational programming facility akin to PROLOG. This it does, but with some differences from PROLOG, notably the use of a breadth-first, rather than depth-first, elaboration of the underlying tree of alternative linear proofs, and the consequent avoidance of explicit backtracking as a control mechanism. It was because of these differences that the facility was called LOGIC rather than PROLOG, which would have been misleading. The name LOGLISP then refers to the combined system: LOGIC + LISP.

It soon became apparent, however, that the main interest of LOGLISP lay rather in its (relatively crude, but genuine) attempt to merge the functional programming style of LISP with the relational programming style of LOGIC and PROLOG. This was done by introducing the notion of 'LISP-transforms' into LOGIC.

The LISP-transform of a simple expression (atomic sentence or term) $E$ of LOGIC is an expression which in many cases is the same as that obtained by applying LISP's EVAL function to $E$.

Thus, the LISP-transform of (PLUS 3 (TIMES 2 6)) is 15. However, the LISP-transform of (PLUS $x$ (TIMES 2 6)) is (PLUS $x$ 12), assuming that $x$ is not defined ('has no value').

It is as though the LISP EVAL function had been modified to be more tolerant of undefined identifiers and to return the symbol or function call itself as its 'value' if it has no value in the usual sense.

Before seeking to unify a goal statement with the heads of appropriate Horn clauses, LOGIC first replaces it by its LISP-transform. This corresponds to PROLOG's concept of first executing the goal statement (if it is executable) but with the difference that it is then sent forward to the unifier for attempted resolutions, rather than being discarded as in PROLOG. Indeed, since real LISP has constructs whose evaluation causes side-effects, these can occur when LOGLISP computes a LISP-transform.

This step of replacing each selected goal by its LISP-transform and then attempting to resolve away the transformed goal has far-reaching consequences.

An obvious and immediate consequence is to provide 'built-in' functions and predicates for LOGIC: any identifier with a LISP definition (whether a system or a user definition) will 'feel' that definition during the LISP-transformation process. In particular a LISP-defined goal sentence such as (LESSP 3 4) would be transformed, in this case to T(RUE), the goal sentence which in LOGIC is always unconditionally provable.

A less obvious consequence is that a goal sentence can contain calls on the LOGIC system itself (since they are LOGIC expressions as well as LISP expressions) such as calls on SETOF or ASSERT. These calls can be nested, so that one can compute, for example, the set of all Jim's cousins who have no sons by LISP-transforming the expression

(SETOF ALL *x* (COUSIN *x* JIM) and (NULL (SETOF ALL *y* (SON *y x*)))

so as to get (say)

(MARY BILL GEORGE).

This rather serendipitous feature of LOGLISP led to the realization that the SETOF construct is quite central in relational programming. Sets (represented in LOGLISP as lists) and relations (sets of tuples) are data objects constructed by deductively evaluating set descriptions and relation descriptions, using sets and relations defined by Horn clauses.

In constructing such sets and relations it is natural in LOGLISP to invoke functions defined in the usual LISP manner and to engineer the overall transaction as a mixture of LISP and LOGIC steps, but there is nothing in the constructions themselves which demands this distinction.

This suggests seeking a more complete merger of the relational and functional paradigms than LOGLISP provides. LOGLISP maintains 'separate but equal' environment management facilities, one for LOGIC variables, which deals with bindings made by unification, and another for LISP variables, which deals with bindings made by assignments and by function calls. The LISP-transformation process is distinguished from the LOGIC process of proving goals by resolution—indeed, these two processes are alternated in a two-phase cycle. The net overall effect is to implement, in this awkward way, a 'LOGLISP-reduction' process, but as a kind of antiphonal duet. It seems clear that there really ought to be only one process, rather than two. Definitions of functions and definitions of relations are not essentially different, and should be invoked in the same way. Variables are variables—there should be just one kind. There is nothing special about 'logical' variables, nor about unification. As will be seen, unification can be viewed as a kind of reduction process, and its steps can be treated in the same way as steps of reduction in general are treated.

## 1. LISP VIEWED AS A REDUCTION SYSTEM

The simplest view one can usefully take of LISP is that it offers the user two facilities: (a) a definition facility; and (b) a deduction facility.

The first of these allows one to define functions by, in effect, asserting new axioms. Each definition is essentially an equation

$$F = \text{lambda}\, X\, B$$

which associates with a symbol $F$ a function described in the notation of the lambda calculus. Here, $X$ is a list of distinct formal variables, while $B$ is the 'body' of the description of the function. Thus one might assert the definition

$$! = \text{lambda}\,(n)\,(\text{if } n = 0 \text{ then } 1 \text{ else } n * (!\,(n-1))$$

of the factorial function by introducing the symbol ! on the left-hand side and describing the function on the right-hand side by means of an expression in which ! occurs. Such recursive definitions are the very stuff of functional programs.

The second facility allows one, in effect, to pose certain kinds of deduction problem and have them solved. One might express these as:

find the expression $V$ in normal form for
which the equation

$$E = V$$

is deducible from the (current set of) axioms.

In LISP one usually thinks of $V$ as the result of applying EVAL to $E$, and calls it the 'value' of $E$. Thus if $E$ is (! 6) one would expect that $V$ would be 720.

Not all expressions are in normal form. The general idea is that if an expression contains one or more subexpressions which can be rewritten in accordance with some definition then it is not yet in normal form and can be further 'reduced' by rewriting one of those subexpressions, or 'redexes', as Curry called them. This 'reduction' process can be kept up for as long as the expression contains such redexes, and in general the replacement of some redexes may well create new redexes, and so on. To be in normal form is, precisely, to contain no redexes.

This view of computation is the reduction point of view which comes with the classical lambda calculus. It automatically entails a 'no error stops' treatment of computation—an expression $E$ always can be transformed, if it is not already in normal form. The point is that, for example, $(x+4)$ is not an error if $x$ is undefined. Instead, it is irreducible, and hence is in normal form. Thus, being in normal form is a relative notion. It depends on the set of definitions which is currently in force. For example, if the definition

$$x = 5$$

is added then the expression $(x + 4)$ is no longer irreducible and reduces to the expression 9.

## 2. Logic VIEWED AS A REDUCTION SYSTEM

The simplest view one can take of Logic (or indeed of 'pure' Prolog) is very similar to the above view of Lisp. Again, two facilities are offered: (a) a definition facility; and (b) a deduction facility.

The first of these allows one to define relations by asserting axioms, called 'positive Horn clauses'. To define $R$ one asserts, in effect, a single equation with a right-hand side which describes a relation by means of a disjunction of simple sentences each corresponding to such a clause:

$$\begin{aligned} R = \text{lambda}\, X\, (\text{OR}\ & (\text{for some } Y_1 : (X = T_1 \text{ and } B_1)) \\ & \quad \vdots \\ & (\text{for some } Y_n : (X = T_n \text{ and } B_n))) \end{aligned}$$

where $X$ is a list of distinct formal variables, $T_i$ is the equally long list of terms which is the argument of $R$ in the $i$th clause, $Y_i$ is the (possibly empty) list of 'local' variables of the $i$th clause, and $B_i$ is a conjunction of atomic sentences comprising the body of the $i$th clause.

In LOGIC and PROLOG one asserts the clauses separately for each $i$ in the form

$$\text{for all } Y_i : R\ T_i \text{ if } B_i.$$

Provided (as Clark [2] argues and as is surely the case) one intends $R$ to hold of a tuple ONLY IF one of these clauses applies, then it is straightforward to see that the conjunction of the separate clauses is equivalent to the single equation.

Another way of thinking about the definition of a relation $R$ by a single equation is that it describes $R$ as a union of relations, namely as

$$\begin{aligned} (\text{UNION}\ & \text{lambda}\, X\ (\text{for some } Y_1 : (X = T_1 \text{ and } B_1)) \\ & \quad \vdots \\ & \text{lambda}\, X\ (\text{for some } Y_n : (X = T_n \text{ and } B_n))). \end{aligned}$$

Although it is not the usual custom to do so, one can read 'lambda $X$' here as 'the set of all $X$ such that', bearing in mind that, after all, a relation is usually construed as a set of tuples.

For example, one can define the APPEND relation in this manner as the union of two relations:

```
APPEND = (UNION lambda (a, b, c) (for some x : a = [] and
                                             b = x and
                                             c = x))
                lambda (a, b, c) (for some x, y, z, w:
                                             a = [x . y] and
                                             b = z and
                                             c = [x . w] and
                                             (APPEND y z w)))
```

the second of which refers recursively to the APPEND relation itself.

The deduction facility of PROLOG or LOGIC is best viewed as one for solving problems of the form:

find the expression $V$ in normal form such that the equation

lambda $X$ (for some $Y$: $A$) = $V$

is provable from the (current set of) axioms.

In other words, find the (normal form description of) the set of all $X$ such that $A$ holds for some $Y$. It is assumed here that the normal form of a set description is one which lists the set's elements, as for example: $\{2, 4, 6\}$. One might note that this customary notation is insignificantly different from

lambda ($x$) ($x = 2$ or $x = 4$ or $x = 6$)

or from its description as the union of the singletons

(UNION (lambda ($x$) $x = 2$) (lambda ($x$) $x = 4$) (lambda ($x$) $x = 6$)).

LOGIC tries to provide such a 'set description' deduction facility with its SETOF function. Different versions of PROLOG vary on this point, but the practice is becoming more and more common to provide such a deductive construct in addition to the basic one, which in effect solves the problem:

find a term $T$ such that the sentence

(lambda $X$ (for some $Y$: $A$) $T$)

is provable from the (current set of) axioms.

The solutions $T$ to this problem are in general not uniquely determined by the condition $A$, and so one speaks of the non-determinacy of the process of finding such a $T$. In fact, however, successive posings of the same problem are not independent, and run through the elements of the set:

lambda $X$ (for some $Y$: $A$)

in some order. Thus, PROLOGS do find the set, but they do it one element at a time; and they do not always offer the service of representing the set by a single expression (or what is the same, as a data object).

So it can be seen that both LISP and LOGIC are essentially in the same business: of accepting definitions in the form of equations and of solving deduction problems by reduction of a given expression to its normal form. Why, then, should they be kept separate from each other? I believe that they need not and should not be.

A system being developed at Syracuse University will now be discussed in which both functional and relational programming merge into a single definition–deduction paradigm. The system is called SUPER (for Syracuse University Parallel Expression Reducer).

SUPER is a reduction system with a repertory of rewrite rules including all those one would expect in a (pure) LISP-like lambda calculus. In particular it will have the usual rule of beta reduction, which calls for the replacement of a redex of the form

(lambda $X$ $B$) $A$

by the expression resulting from substituting $A$ for (free) $X$ throughout $B$. The constants

EVERY, SOME

are added to the language so as to provide the logical quantifiers via the constructions

for all $X$: $A$ = (EVERY (lambda $X$ $A$))

for some $X$: $A$ = (SOME (lambda $X$ $A$))

which go back to Church's language [3] for the simple theory of types. SUPER will be based on that system (which has been studied by Henkin [4] who gave a completeness result for it, and which recently was used by Andrews [5] as the formalism for a higher-order theorem proving system). Thus SUPER has only two syntactic constructs: application (of one expression as function, to another expression as argument) and abstraction. Its expressive power comes from a suitable collection of constants: TRUE, FALSE, NOT, AND, OR, IF, IFF, for the Boolean combinations of elementary logic; CAR, CDR, CONS, ATOM, EQ, COND, NIL, etc., for the LISP-like symbolic apparatus; PLUS, TIMES, etc., together with suitable numerals, for arithmetic. All of these constants will require appropriate reduction rules to give them operational meaning, and all of them are familiar and straightforward with the exception perhaps of those dealing with the SUPER version of unification. These will be discussed next.

Let us follow a few of the transformations necessary to compute the normal form of the following set description:

$$\text{lambda } (p, q) \ (\text{APPEND } p \ q \ [1\,2\,3]) \tag{1}$$

which we can see, intuitively, is

$$\begin{aligned}&\{([], [1\,2\,3])\\ &([1], [2\,3])\\ &([1\,2], [3])\\ &([1\,2\,3], [])\}.\end{aligned} \tag{1'}$$

Here and henceforth the convention for list notation whereby, for example, the list [1 . [2 . [3 . []]]] is written more readably as [1 2 3] is used.

The expansion rule takes the definition of APPEND as the union of two simple relations and rewrites (1) as the union of two relations

$$\begin{aligned}(\text{UNION lambda } (p, q) \ (\text{for some } x{:}\ & p = [] \text{ and}\\ & q = x \text{ and}\\ & [1\,2\,3] = \underset{\smile}{x})\\ \text{lambda } (p, q) \ (\text{for some } x,y,z,w{:}\ & p = [x\,.\,y] \text{ and}\\ & q = z \quad \text{and}\\ & [1\,2\,3] = [x\,.\,w] \text{ and}\\ & (\text{APPEND } y\ z\ w)))\end{aligned} \tag{2}$$

(In applying the expansion rule it is necessary to pay attention to the question of clashes of bound variables and to take precautions of the usual kind, namely, to change the local variables of the set descriptions, if necessary, before replacing the formal variables by actual terms of the redex.)

In the second component of the union expression there is an expansion redex, but there are now two redexes of another kind, one in the first and one in the second component of the union. These call for applications of the contraction rule. According to this rule a local variable can be dropped from the quantifier prefix of the body of a simple relation expression provided that one of the conjuncts in the body is an equation between that variable and some term which does not contain it. In the first component above the variable is $x$ and the term is [1 2 3]. In the second component the variable is $z$ and the term is $q$. In addition to dropping the variable from the prefix all of its occurrences in the body must be replaced by the term. So in the above $x$ is replaced throughout the body of the first component by [1 2 3 ], and $z$ throughout the body of the second component by $q$. Finally, the trivial equation thus created is dropped. In the first component this is the equation [1 2 3] = [1 2 3]. In the second component it is $q = q$. The resulting expression is

```
(UNION lambda (p, q) (p = [] and q = [1 2 3])
        lambda (p, q) (for some x, y, w: p = [x . y]  and
                                         [1 2 3]  = [x . w] and        (3)
                                         (APPEND y q w)))).
```

It should be noted that the contraction rule applies only to redexes of the particular kind described. The form in general is

lambda $A$ (for some $B$: $C_1$ and $\cdots$ and $C_m$ and $(V = E)$ and $D_1$ and $\cdots$ and $D_n$)

where the list $B$ of local variables contains the variable $V$ and $E$ is a term not containing $V$. (The equation can also be $E = V$.) The redex is replaced by

lambda $A$ (for some $B'$: ($C_1$ and $\cdots$ and $C_m$ and $D_1$ and $\cdots$ and $D_n$)$\{E/V\}$)

where $B'$ is the list $B$ with $V$ omitted. That is, we replace $V$ by $E$ throughout the conjunction, and drop the trivial equation $E = E$ thus created.

Notice that in this example contraction is applied simultaneously to two different redexes. This is a small example of the way in which reduction can be done in parallel.

The next step illustrates the decomposition rule. This states that a redex of the form

lambda $A$ (for some $B$: ($C$ and $[P . Q] = [R . S]$ and $D$))

where $C$ and $D$ are both simple conjunctions (possible empty), may be replaced by

lambda $A$ (for some $B$: ($C$ and $P = R$ and $Q = S$ and $D$)).

All that this rule is saying is that an equation between two dotted pairs is equivalent to two equations between their respective heads and tails. Applying decomposition to (3) yields

```
(UNION lambda (p, q) (p = [] and q = [1 2 3])
        lambda (p, q) (for some x,y,w: p = [x . y]
                                          1 = x                        (4)
                                       [2 3] = w and
                                        (APPEND y q w)))
```

thus creating two new equations which permit contraction to be applied twice more, after which we have

$$\begin{aligned}&(\text{UNION lambda}\,(p, q)\ (p=[\,]\text{ and }q=[1\,2\,3])\\ &\qquad \text{lambda}\,(p, q)\ (\text{for some }y{:}\,p=[1\,.\,y] \qquad (5)\\ &\qquad\qquad (\text{APPEND}\ y\ q\ [2\,3])))\end{aligned}$$

and we must take stock of what has been happening to the original set description (1) as it is step-by-step being transformed into the description (1′).

In (5) already a singleton set containing the first couple has emerged. The other three elements have yet to emerge from the description of the rest of the set. However, this description has been partially developed, and now intuitively reads 'the set of all couples of lists, the first of which starts with 1 and has a tail which, when appended to the second, yields the list [2 3]'. We can intuitively see that this is the set

$$\{([1], [2\,3]), ([1\,2], [3]), ([1\,2\,3], [\,])\}.$$

Further applications of the three rules expansion (using the definition of APPEND), contraction and decomposition will carry (5) step-by-step nearer to the union of singleton sets which is the required normal form representing (1′). Only one further rule is required to complete the overall transformation. This is the failure rule which states that a redex of the form

$$(\text{UNION}\ S_1 \cdots S_m\ (\text{lambda}\ A\ (\text{for some}\ B{:}\ C))\ T_1 \cdots T_n)$$

where $C$ is a conjunction of simple sentences one of which is obviously false, may be replaced by the expression

$$(\text{UNION}\ S_1 \cdots S_m\ T_1 \cdots T_n).$$

The intuitive justification of the rule is that the deleted component describes the empty set.

Some examples of 'obviously false' simple sentences are:

$$\text{FALSE},\ [\,]=[x\,.\,y],\ 2=3.$$

The failure rule is so named because it corresponds to the occasions in the unification process when an attempt to unify two expressions fails.

Let us skip forward to the point where the failure rule comes into play in our example. The following description is reached after applying an expansion:

```
(UNION lambda (p, q) p = [] and q = [1 2 3]
       lambda (p, q) p = [1] and q = [2 3]
       lambda (p, q) p = [1 2] and q = [3]
       lambda (p, q) p = [1 2 3] and q = []
       lambda (p, q) (for some x, y, z, w:
                           p = [1 2 3 . y]  and
                           y = [x . z]      and
                           [] = [x . w]     and
                           (APPEND y q w)))
```

whose final component contains the impossible equation $[] = [x \,.\, w]$. It also contains a further recursive call on the APPEND definition, and hence is an expansion redex. In addition, by virtue of containing the equation for the local variable $y$ it is a contraction redex. However, the rewriting of the entire UNION expression by the failure rule simply drops these redexes and thus would be the most advantageous choice.

The final expression is then (ensugaring the singletons):

```
(UNION {([], [1 2 3])}
       {([1], [2 3])}
       {([1 2], [3])}
       {([1 2 3], [])})
```

which can be further ensugared to the form of (1′) if desired.

The overall computation sketched in this example corresponds, in a reduction setting, to the complete exploration of the tree of alternative Horn-clause resolution deductions which PROLOG or LOGIC would perform in response to a request to find the set of all $(p, q)$ such that (APPEND $p$ $q$ [1 2 3]). The expansions correspond to the invocations of the two clauses of the APPEND definition; the contractions to the successive bindings made by the unification process in attempting to unify a selected goal with a clause head; the decompositions to the recursive calls to the unification algorithm when two dotted pairs are to be unified; and the failures to the moments when the unification algorithm encounters an impossible combination.

## 3. SCOPE OF SUPER

Although the underlying language of SUPER is the simple theory of types, sometimes also called the predicate calculus of order omega, at present the experiment has not gone far enough to know whether the repertory of rules can be usefully extended to cover higher-order unification. Huet [6] and Pietrzykowski and Jensen [7] have given unification algorithms for the typed lambda calculus, but the computational problems are much more complex than in the first order case.

The objectives so far have been limited to reorganizing the present stock of ideas about first-order relational programming so that at least the same capability one has in LOGIC can be reproduced in the reduction setting. The next goal is to investigate the feasibility of implementing SUPER, as it presently exists, in a parallel reduction architecture. Klaus Berkling is currently designing a second version of his well-known GMD Reduction Machine [8] which will embody the expansion, contraction, decomposition and failure rules as well as a full set of rules suitable for a LISP-like functional language. This machine will have a multiprocessor architecture and its design is at present under way.

## 4. FUTURE WORK

It would, of course, be very interesting to extend these rules beyond the first-order unification level. Another line of investigation is to see how far, if at all, one can push the combinator approach which in recent years has been so well exploited by Turner [9]. The main problem seems to be that transformation of a lambda abstraction to pure combinator form disarranges the syntactic structure of the original so much that the unification analysis cannot be carried out. In the present system this comes out in the contraction rule: in order to apply the rule one has to identify the term which will be substituted for the variable being eliminated. This is easy enough in the original expression, but after transformation into pure combinator form there are no longer any variables and in particular no equations between variables and terms. There seems therefore to be no way to identify the term. So the question is: what corresponds to the unification process after all variables have been transformed away?

It would be useful to know more about the role played by the typing of expressions in the SUPER language. Huet's higher-order unification algorithm makes crucial use of the types of expressions at certain stages, but none of the rules considered in this discussion do, as witnessed by the fact that types have scarcely been mentioned.

Finally, it should be said that aspirations do not presently extend to building a complete proof procedure for SUPER, although this is not, in view of Henkin's result cited earlier [4], out of the question. Rather something like a higher-order Horn clause resolution theorem prover is sought in which it will be possible to do logic computations à la Kowalski [10] but without the restriction to first order. Related work is in progress at Imperial College by Darlington and his group [11] and at Cambridge University by Paulson [12]. The point is to retain, if possible, the directed purposefulness of a computation process, and not to slide back into the world of mere searching.

## REFERENCES

1. Robinson, J. A. and Sibert, E. E. (1982) LOGLISP: an alternative to PROLOG. In *Machine intelligence 10* (eds J. E. Hayes, D. Michie, and Y.-H. Pao) pp. 399–419. Ellis Horwood, Chichester and New York, Halsted Press.
2. Clark, K. L. Negation as failure. In *Logic and databases* (eds H. Gallaire and J. Minker) pp. 293–294. Plenum Press, New York.
3. Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic* **5**, 56–68.
4. Henkin, L. (1980) Completeness in the theory of types. *J. Symbolic Logic* **15**, 81–91.
5. Andrews, P. B., Miller, D. A., Cohen, E. L., and Pfenning, F. (1984) Automating higher order logic. In *Automated theorem proving: after 25 years* (eds W. W. Bledsoe and D. Loveland) *Contemporary mathematics*, Vol. 29, pp. 169–92. American Mathematical Society.
6. Huet, G. (1975) A unification algorithm for typed lambda calculus. *Theoretical Computer Science* **1**, 27–57.
7. Pietrzykowski, T. and Jensen, D. A complete mechanization of omega-order type theory . *ACM National Conference* 1972, Vol. 1, pp. 82–92.
8. Berkling, K. J. (1976) Réduction languages for reduction machines. *Gesellschaft für Mathematik und Datenverarbeitung* **957**, Bonn.
9. Turner, D. A. (1979) A new implementation technique for applicative languages. *Software Practice and Experience* **9**, 31–49.
10. Kowalski, R. A. (1979) *Logic for problem solving.* North Holland, Amsterdam.
11. Darlington, J., Field, A. J., and Pull, H. (1985) The unification of functional and logic languages. Report DOC 85/3. Imperial College, London.
12. Paulson, L. C. (1985) Natural deduction theorem proving via higher order resolution. Technical report No. 67. Computer Laboratory, Cambridge University.

# 4
# Concurrent Computer Architecture for Unification Operations

J. V. Oldfield
Department of Electrical and Computer Engineering, Syracuse University, USA
C. D. Stormon
Centre for Computer Applications and Software Engineering, Syracuse University, USA

## 1. INTRODUCTION

There is considerable and growing interest in the development and application of logic programming languages such as PROLOG and LOGLISP. At the same time it is increasingly clear that conventional computers are ill-adapted to the execution of logic programs. The advent of LISP machines inspired by the MIT CADR has improved the situation for LOGLISP, but considerable further improvement should be possible. All logic programming systems are based on unification. This is a process for calculating the most general way in which a given environment of variable bindings can be extended (by adding further bindings if necessary) to one in which two given symbolic expressions are identical, or for detecting the impossibility of doing so (as the case may be). In the first case, the extended environment is retained. Otherwise the extension must be removed.

Since unification accounts for a great proportion of the activity of logic programming systems, present-day systems such as LOGLISP incorporate sophisticated techniques based on hash-coding and stack structures to speed-up the process. The advent of LISP machines such as the LMI Lambda and Symbolics 3600 has improved the situation considerably over implementations on conventional computers such as DEC KL-10, but none of the hardware is designed specifically to carry out the unification processes efficiently. Application of logic programming techniques in potentially significant areas such as large-scale expert systems is presently limited by the speed of available computers, and so it is appropriate to look for further improvement by special purpose computer hardware.

Several years ago, Lien (1981) published a design for a unification chip as part of an M.Sc. thesis. Although the chip was never fabricated this work showed that a special-purpose unification processor was feasible.

The design was a direct implementation of an early published algorithm for unification (Robinson, 1971). More recently developed algorithms (Morris, 1978) are much more efficient and are in wide use. Before describing the unification process, some definitions are appropriate.

An expression is either a *variable,* or a *constant,* or a (*'dotted'*) *pair* ($C.D.$) of expressions in which $C$ is its *head* and $D$ is its *tail*. An environment is a collection of pairs, each of which has a distinct variable as its head. An environment $E'$ is an *extension* of an environment $E$ if every pair in $E$ is also in $E'$.

The *realization* of an expression $X$ in an environment $Y$ is:

- $X$, *if $X$ is a constant*:
- the realization of $Z$ in $Y$, *if $X$ is a variable and the pair* ($X.Z$) *is in* $Y$;
- $X$, *if $X$ is a variable and there is no pair in $Y$ whose head is $X$*;
- the pair whose head is the realization of $C$ in $Y$ and whose tail is the realization of $D$ in $Y$, if *$X$ is the pair* ($C.D$).

The unification task can be described as follows. We are given two expressions $A$ and $B$ together with an *environment E*. We are to find (if one exists) a most general extension $E'$ of $E$ such that the realizations of $A$ and of $B$ in $E$ are identical; or to show that none exists, as the case may be.

The following algorithm solves this problem.

Let (UNIFY $A$ $B$ $E$) be:

- 'impossible', if $E$ is :impossible';
- $E$, if $A = B$;
- (UNIFY $A'$ $B$ $E$), if $A$ is a variable and ($A.A'$) is in $E$;
- (UNIFY $A$ $B'$ $E$), if $B$ is a variable and ($B.B'$) is in $E$;
- (UNIFY $A1$ $B1$ (UNIFY $A2$ $B2$ $E$)) if $A$ is the pair ($A1.A2$) and $B$ is the pair ($B1.B2$);
- $E$ plus the pair ($A.B$), if $A$ is a variable and no pair in $E$ has $A$ as head;
- $E$ plus the pair ($B.A$), if $B$ is a variable and no pair in $E$ has $B$ as head;
- 'impossible', otherwise.

In 1982 we began a preliminary study for a computer system which would incorporate the latest thinking on unification algorithms, along with concurrent computer operation and the incorporation of custom VLSI chips where appropriate. This led to an overall architectural scheme (Greene, personal communication) which will now be described.

## 2. THE SYRACUSE UNIFICATION MACHINE

The Syracuse Unification Machine (SUM) is designed as an experimental co-processor for an existing host machine running LOGLISP. SUM assists the host in performing unifications and keeping track of the variable-to-expression bindings which are built up during the execution of a logic program.

### 2.1. Scope of operation

Many unifications are required in the course of running a typical logic program. Some of these unifications are of a degenerate nature in that they will fail immediately. These are the cases where two objects of incompatible structure (e.g. a pair and an atom) are to be unified. These are the types of unifications that can often be 'compiled away' in typical logic program compilers. Another simple case occurs when two constants are the objects to be unified. A unification of this type will succeed if the constants are identical and fail otherwise. Both of these types of unification require only the simple bitwise comparison of words in memory and thus are judged to be best handled by the host machine because of the communication overhead involved in passing them to a co-processor.

The great majority of unifications required in the execution of a logic program fall into one of the two remaining categories: the unification of a pair with a pair, and the unification of any expression with a variable. In order to unify a pair with another pair, it is necessary to unify the heads of the pairs and, recursively, the tails of the pairs. Since SUM is currently designed to deal with pointers to pairs, but not have direct access to their heads and tails, the host machine must look these up and pass them to SUM when appropriate. Future versions of SUM might make use of direct memory access techniques to allow the co-processor to follow the pointers to the heads and tails.

Thus it is only when at least one of the two expressions to be unified is a variable that SUM becomes involved. When the execution of a logic program requires that a variable be unified with some expression, the host passes the task directly to SUM. This case is potentially the most complicated, and the one where special-purpose hardware is most likely to improve performance.

### 2.2. Principles of operation

SUM consists of four major types of units: the Communication Agent, the Binding Controller, the Binding Agent, and the Analysis Agent. Each of these units is designed to operate concurrently to increase efficiency. The proposed configuration of SUM is depicted in Figure 1.

When operation is initiated, the host machine loads an environment of bindings (possibly empty) into SUM. SUM maintains and extends this environment as required while performing unification operations. The extended environment may be reported by SUM at the request of the host machine. During its operation SUM receives as input from the host machine a stream of tasks of the form,

(UNIFY $V\,X$)

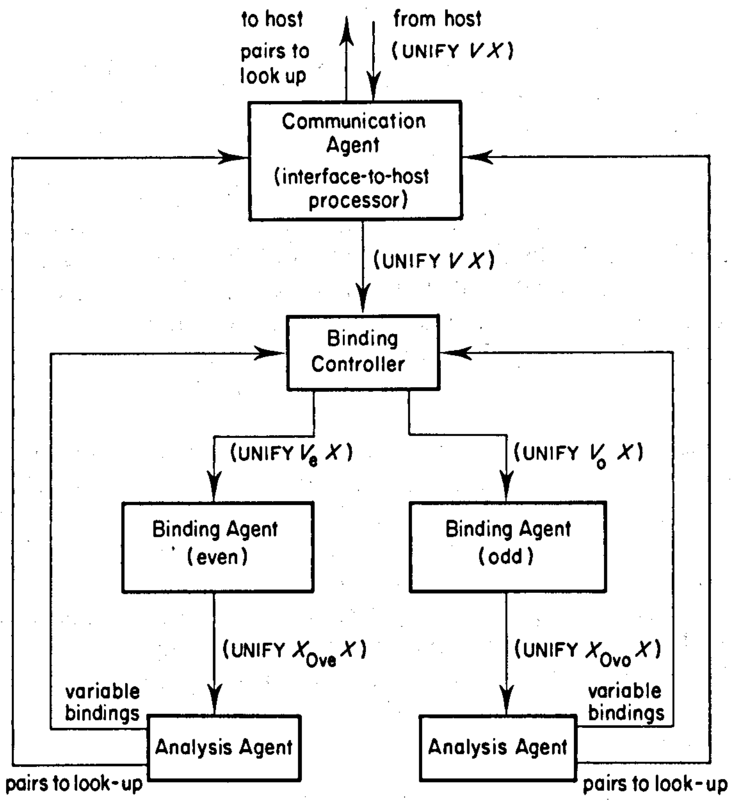


Figure 1. The Syracuse Unification Machine (SUM).

where $V$ is a variable and $X$ may be a variable, a pointer to a pair, or an atom.

### 2.2.1. *The Communication Agent*

The Communication Agent provides a two-way communication interface between SUM and the host machine. Items from the host are transferred to the Binding Controller while items which originate in SUM are passed to the host.

### 2.2.2. *The Binding Controller*

The Binding controller places unification tasks of the above form on a FIFO stack. The Binding Controller has the job of assigning each task to the appropriate Binding Agent. In the current design, there are two Binding Agents (there may be one, two or many). Part of each variable identifier from the host is an index which identifies the variable's location in the search tree. If $X$ and $V$ are both variables, the Binding Controller will order $V$ and $X$ so that the variable with higher index comes first. This is to ensure that bindings are never cyclic. With two Binding Agents, the Binding Controller simply assigns the task whose left side has even index to one Binding Agent while those with odd index are sent to the other.

## 2.2.3. *The Binding Agent*

The Binding Agent is the heart of SUM; its design is discussed in Section 3. A flowchart describing the operation of the Binding Agent is shown in Figure 2. The Binding Agent contains a stack of variable-to-expression

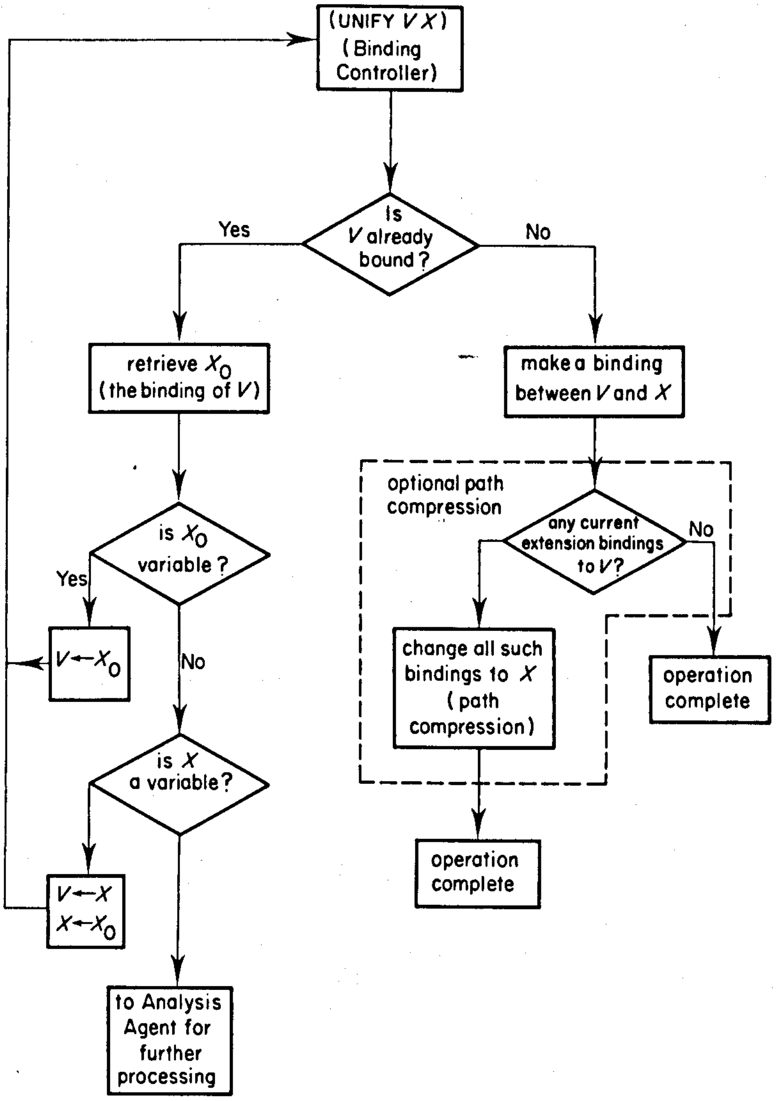

(UNIFY V X)
(Binding Controller)
Is V already bound?
Yes
No
retrieve $X_0$ (the binding of V)
make a binding between V and X
optional path compression
any current extension bindings to V?
No
is $X_0$ variable?
Yes
No
$V \leftarrow X_0$
change all such bindings to X (path compression)
operation complete
is X a variable?
$V \leftarrow X$
$X \leftarrow X_0$
operation complete
to Analysis Agent for further processing

Figure 2. Flowchart of binding operations.

bindings which are made in the course of executing unification tasks. When a new task of the form (UNIFY *V X*) comes in, the Binding Agent checks to see if the variable *V* has already been bound to something. If a search of the stack of bindings shows no entry for variable *V*, the new binding of *V* to *X* is pushed onto the stack, and the unification step is completed.

If there is already an entry for the variable *V* in the stack of bindings, the only way that the operation (UNIFY *V X*) can succeed is if the expression *X* can be unified with the expression to which the variable was already bound (say *X*0). Thus the Binding Agent retrieves the binding of *V* which is *X*0 and produces a task which has the form,

(UNIFY *X X*0)

which is passed on to the Analysis Agent. The existing binding of *V* to *X*0 is left in place.

### 2.2.4 *The Analysis Agent*

The design of the Analysis Agent is described in Section 4. The Analysis Agent performs a case analysis of the unification tasks it receives from the Binding Agent. If one of the expressions is a variable, the task is passed to the Binding Controller. If the expressions are of incompatible structure the Analysis Agent signals failure of unification. If the expressions are both simple atoms, the Analysis Agent compares them and signals failure if they are not identical.

If the expressions are pairs or if they are pointers to complex atoms the Analysis Agent passes the unification task to the Communication Agent and thus back to the host machine. In the case of pairs, the host machine will return unification tasks for the heads and tails of these pairs.

### 2.2.5 *Success or failure*

Unification can fail in the Analysis Agent or in the host machine. When a failure is signalled, all the units can flush the work they are currently doing. In addition, the current extension to the environment (all the bindings that were added as a result of performing that unification) are thrown away. In this way the machine can operate in the familiar 'backtracking' mode used in PROLOG systems.

The success of a unification task may result in bindings being added to the environment, but there is no need to signal other units or the host of each success. When ultimately there are no more unifications to be performed, and no failures have been signalled, the Binding Agents can be said to contain the 'answer' to the query given the logic program. The 'answer' consists of substitutions for the variables in the query which make it a true statement. These substitutions may be passed back to the host (at the request of the host) for reporting to the user, or for further processing.

## 3. THE BINDING AGENT

The Binding Agent holds the binding environment in a specially designed memory with content addressable features. At present we envisage a total of two Binding Agents, with the work divided on the basis of index parity, but more (or fewer) may be necessary. Unifications are carried out as part of a higher level *resolution* process, and at each successful unification the binding environment will be extended. Conversely, if unification fails, the environment must be contracted to correspond to the last successful step. In consequence, we have organized the Binding Agent memory as a stack.

When presented with a binding task, the Binding Agent must determine if the variable is already bound to an expression. We have developed a custom VLSI chip which allows simultaneous look-up on 128 rows of 17-bit-wide variable identifiers in 100–150 ns. A full-scale Binding Agent will employ 32 such chips and so hold a maximum of 4096 entries. The chip uses a nine-transistor static CAM cell which is derived from the well known six-transistor static RAM cell plus three transistors for comparison between the cell contents and the search bit. A novel feature is the use of a mask register which avoids the space taken by an address decoder and which serves to indicate the extent of the current stack and also the next free location. A prototype version, with eight rows of three bits but otherwise complete, was designed by S.-H. Kuang and successfully tested after fabrication by the MOSIS Fast Turnaround Fabrication Facility.

If the variable presented is not already bound, the corresponding expression is written to the RAM memory of the Binding Agent. The organization is conventional and uses fast static RAM chips in 4K × 4 format. If, however, the variable is already bound, the stored expression must be unified with the newly arrived expression. This requires a read cycle in the RAM followed by transmission to the Analysis Agent.

Figure 3 shows the general organization of the Binding Agent. Each Binding Agent has its own controller, which maintains a stack of environment extensions pointers. In the event of unification failure, the controller works out the number of shifts to be applied to the mask register and so remove recent extensions from the stack. We have found that shifting may take place at up to 15 MHz, and so this housekeeping operation is executed very quickly.

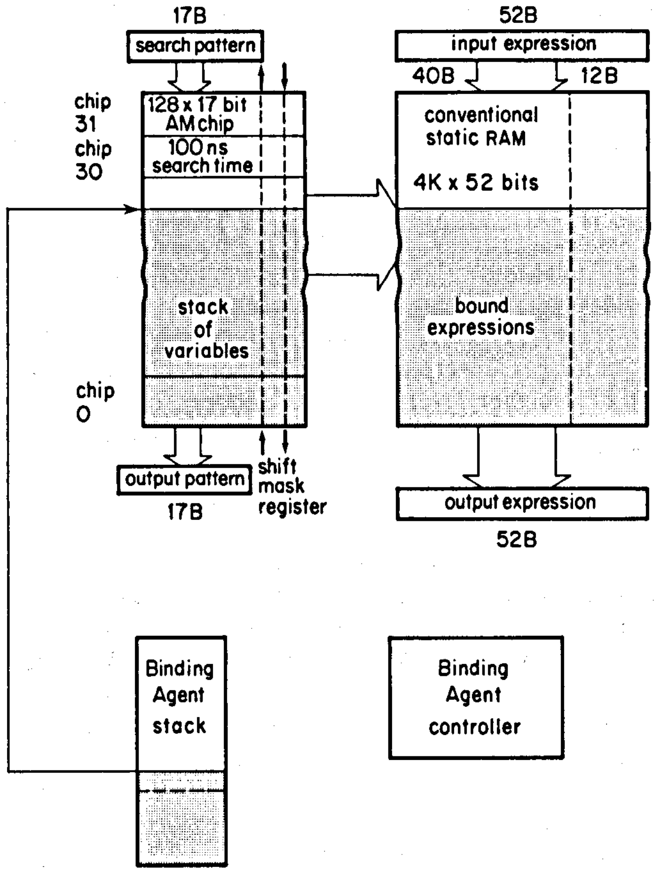


Figure 3. Structure of the Binding Agent.

## 4. THE ANALYSIS AGENT

A preliminary design for the Analysis Agent was carried out by V. Venkataraman in 1984 (Figure 4). Because of pin-out limitations and the wide data paths required, it is necessary to break down data into sequences of bytes. These are passed in pairs through a front end comparator which reports if bytes match or not. By careful ordering we can determine if unification is feasible and route the data to either the Binding Agents or to the Communication Agent. The design uses dynamic shift registers to hold the data prior to dispatch, and a programmable logic array serves as a controller.

## 5. STATUS OF THE SUM PROJECT

At present work is proceeding on a simulation of SUM using the OCCAM language. OCCAM allows the direct modelling of concurrent processes

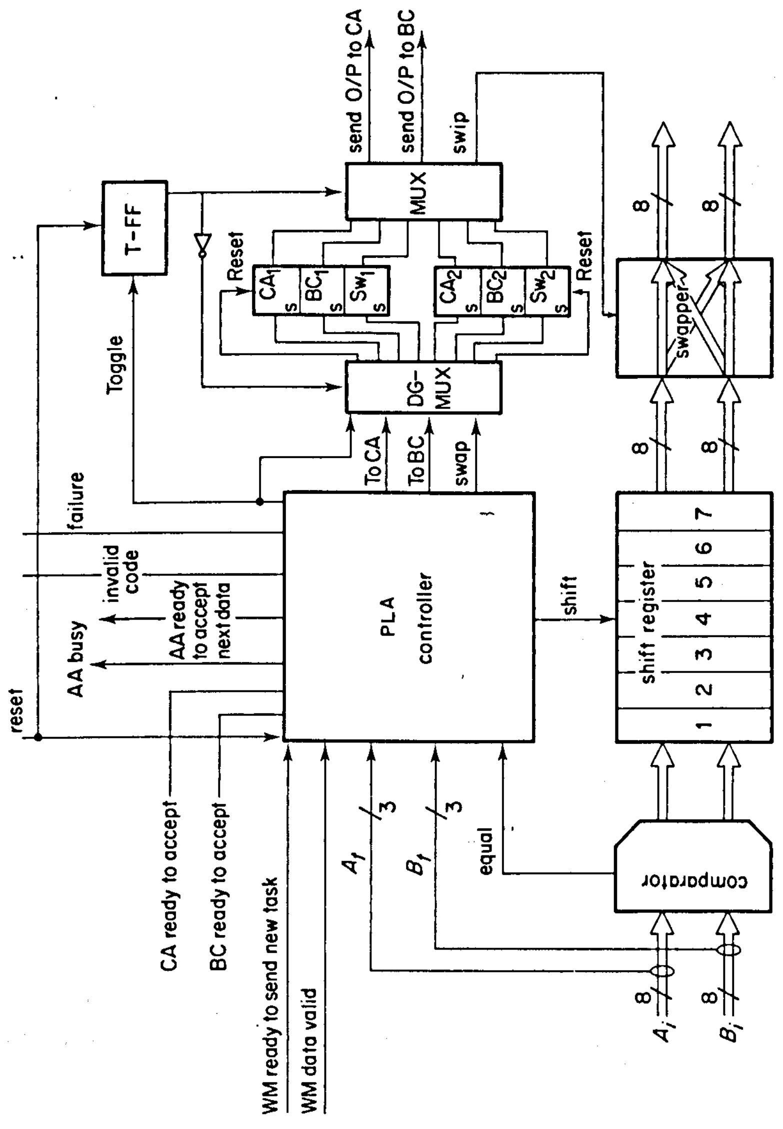


Figure 4. Analysis Agent pipeline architecture.

through the use of PARallel and ALTernate constructors (Hoare, 1984). When complete, this simulation will provide information with regard to the level of performance achievable with SUM, and assist us in evaluating the design. We are considering implementing SUM using the Inmos transputer, which executes OCCAM (Barron, 1978). In that case, the simulation program would be the basis for the actual SUM system configuration.

Our implementation plans include interfacing a single transputer to a host machine running LOGLISP (candidate machines include the LMI Lambda, the Symbolics 3600, and the IBM PC AT). This would enable us to run our OCCAM model of SUM in connection with a working LOGLISP system. As the hardware Binding Agents and Analysis Agents are completed, they can be interfaced to this transputer system for testing. Eventually, the entire SUM processor will be constructed with the original transputer system performing the functions of the Communication Agent.

Our preliminary research indicates that SUM will provide approximately 100:1 average performance improvement for binding operations over typical software methods. The overall logic programming system performance increase will be much more modest however. The purpose of the SUM project is to investigate the feasibility and utility of special-purpose unification hardware using modern integrated circuit technology. With advances in VLSI technologies, it should be possible to implement an entire Unification Machine on a single chip. This will make it practical to have several complete LOGLISP or PROLOG machines operating concurrently on different parts of the logic program search tree.

The attraction of the Unification co-processor in this arrangement is that it is a general-purpose processing element which makes no assumptions about the execution order of logic programs. Therefore, it can be used in a pure depth-first scheme with backtracking on a unit processor (as in PROLOG), in a heuristic search scheme (as in LOGLISP currently on unit processors), and in any number of multi-processing schemes. In addition, the data structures used in SUM are quite generic which means that the SUM architecture should be applicable to almost any logic programming system, not just PROLOG and LOGLISP. This last fact is of special interest to us, since work on new logic and functional programming systems is also going on here at Syracuse University (Greene, personal communication).

**Acknowledgements**

The authors wish to thank their colleagues for many useful discussions, and in particular J. A. Robinson, E. E. Sibert, K. J. Greene, H. Schlütter, and S.-H. Kuang. They also are grateful for the support of IBM Corporation by way of a Departmental Grant, and to the CASE Center of Syracuse University.

## BIBLIOGRAPHY AND REFERENCES

Barron, I. (1978) The transputer. In *The microprocessor and its applications* (ed. D. Aspinall). Cambridge University Press, Cambridge.

Berkling, K. (1983) Experiences with integrating parts of the GMD-Reduction Language Machine. In *VLSI architecture* (eds B. Randell and J. Treleaven). Prentice-Hall, Englewood Cliffs, N. J.

Hoare, C. A. R. *OCCAM programming manual.* Prentice-Hall, Englewood Cliffs, N.J.

Lien, S.-L. C. (1981) Towards a Theorem Proving Architecture, Tech. Report 4653. Computer Science Department, California Institute of Technology.

Luner, D. L. (1984) A concurrent procedure for unification. M.S. Dissertation, Syracuse University.

Morris, F. L. (1978) On List Structures and their use in the programming of Unification. Tech. Report CIS 4–78. School of Computer and Information Science, Syracuse University.

Oldfield, J. V. and Kuang, S.-H. (1984) An associative stack chip for the Syracuse Unification Machine. Tech. Report, CASE Center, Syracuse University.

Robinson, J. A. (1971) Computational logic: The unification computation. In *Machine Intelligence 6* (eds B. Meltzer and D. Michie), pp. 63–72. Edinburgh University Press, Edinburgh.

Robinson, P. The SUM: an AI co-processor. *Byte* **10**(6), 169–80.

# DEDUCTIVE PROBLEM-SOLVING AND PROOF

5

# Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic

R. S. Boyer and J S. Moore
Institute for Computing Science and Computer Applications, University of Texas at Austin, USA

**Abstract**
We discuss the problem of incorporating into a heuristic theorem prover a decision procedure for a fragment of the logic. An obvious goal when incorporating such a procedure is to reduce the search space explored by the heuristic component of the system, as would be achieved by eliminating from the system's data base some explicitly stated axioms. For example, if a decision procedure for linear inequalities is added, one would hope to eliminate the explicit consideration of the transitivity axioms. However, the decision procedure must then be used in all the ways the eliminated axioms might have been. The difficulty of achieving this degree of integration is more dependent upon the complexity of the heuristic component than upon that of the decision procedure. The view of the decision procedure as a 'black box' is frequently destroyed by the need to pass large amounts of search strategic information back and forth between the two components. Finally, the efficiency of the decision procedure may be virtually irrelevant; the efficiency of the final system may depend most heavily on how easy it is to communicate between the two components. This paper is a case study of how we integrated a linear arithmetic procedure into a heuristic theorem prover. By *linear arithmetic* here we mean the decidable subset of number theory dealing with universally quantified formulas composed of the logical connectives, the identity relation, the Peano 'less than' relation, the Peano addition and subtraction functions, Peano constants, and variables taking on natural values. We describe our system as it originally stood, and then describe chronologically the evolution of our linear arithmetic procedure and its interface to the heuristic theorem prover. We also provide a detailed description of our final linear arithmetic procedure and the use we make of it. This description graphically illustrates the difference between a stand-alone decision procedure and one that is of use to a more powerful theorem prover.

## 1. INTRODUCTION

Decision procedures, alone or in co-operation with other decision procedures, are fast and predictable but often too limited to be of general use. On the other hand, today's heuristic theorem provers are capable of producing proofs of fairly deep theorems, but are generally so slow and unpredictable that few users have the patience and knowledge to use them effectively. It is generally agreed that when practical theorem provers are finally available they will contain both heuristic components and many decision procedures.

This paper is a case study of how we integrated into a heuristic theorem prover a linear arithmetic procedure for the natural numbers based on a decision procedure for the rationals. By *linear arithmetic* here we mean the decidable subset of number theory dealing with universally quantified formulas composed of the logical connectives, the identity relation, the Peano 'less than' relation, the Peano addition and subtraction functions, Peano constants, and variables taking on natural values. We built in linear arithmetic primarily to eliminate from the heuristic theorem prover's search space the huge number of often irrelevant deductions arising from such theorems as the transitivity of the 'less than' relation.

This paper can be divided up into three distinct phases. The first, represented by Sections 2–4, argues that it is necessary to combine decision procedures and heuristic theorem provers. During the first phase we also give some necessary background material on our heuristic theorem prover and what we mean by 'linear arithmetic procedures'. The second phase, Sections 5–7, describes chronologically our attempts to incorporate linear arithmetic into our theorem prover. In this phase we cite examples from program verification applications that show the inadequacy of our early integration strategies and that illustrate and motivate our final scheme. The third phase of the paper, Section 8, gives a precise and detailed description of the current linear arithmetic procedure and how it is used by the rest of the theorem prover. The final scheme is so elaborate that reading it in isolation would prompt many readers to ask such questions as 'why is it necessary to know which literals contributed to the deduction?' or 'why didn't the authors use this simpler scheme?'. Despite the tedium of this description we regard Section 8 as the high point of the paper because it makes clear the distinction between a stand-alone decision procedure and one that is useful to a larger system. The last two sections of the paper give some statistics supporting our contention that the efficiency of the stand-alone decision procedure is often irrelevant and a summary of our conclusions.

We believe this report will be useful to those designing decision procedures intended for eventual integration into larger systems. We identify many requirements for such procedures that are not obvious when the procedures are considered in isolation.

For example, much work on linear arithmetic procedures (e.g. that of Nelson and Oppen, 1979, and Shostak, 1979) focuses on universally quantified formulas either with no function symbols (other than sum and difference) or with only uninterpreted function symbols. But interpreted function symbols play a key role in many theorem-proving applications. In particular, they are crucial in what is perhaps the most active application of mechanical theorem provers today: the verification of properties of computer programs. The mathematical specification of new programs frequently involves 'new' mathematical functions (e.g. 'the number of non-0 elements among the first $N$ elements of $A$'). Furthermore, these functions very frequently have important numeric relations to one another (e.g. if $N<M$ then the number of non-0 elements among the first $N$ elements of $A$ is less than or equal to the number among the first $M$ elements). Unless provision is made for one's arithmetic procedure to take into consideration the numeric properties of interpreted symbols, the heuristic theorem prover must deal explicitly with such explosive theorems as the transitivity of 'less than' and the primary advantage in having an arithmetic decision procedure is lost.

The work reported here deals with interpreted function symbols: our linear arithmetic procedure contains heuristics for instantiating and using axioms or lemmas about arbitrary function symbols. For example, by appealing to the lemma that the minimum element, $\textsc{min}(A)$, of a sequence is less than or equal to the maximum element, $\textsc{max}(A)$, our linear arithmetic procedure proves:

$$L \le \textsc{min}(A) \wedge 0 < K \rightarrow L < \textsc{max}(A) + K.$$

The use of universally quantified axioms or theorems in the linear procedure is very similar to the admission of universally quantified hypotheses in the formulas being proved. Thus, our work is similar in spirit to the work of Bledsoe and Hines (1980) in which arbitrary quantification is permitted. However, we make no completeness claims about our heuristics.

## 2. BACKGROUND

Our theorem prover deals with a quantifier-free, first-order logic. In addition to modus ponens, instantiation, and substitution of equals for equals, the logic provides for the axiomatic introduction of new 'types' of inductively constructed objects (e.g. the natural numbers, sequences, graphs), the user definition of new mathematical functions (e.g. prime, permutation, path), and proof by induction on well-founded relations. The logic is described precisely in Chap. III of Boyer and Moore (1979).

Our theorem prover as it stood before we incorporated any linear arithmetic is described in Chaps V–XV of Boyer and Moore (1979). The theorem prover consists of an *ad hoc* collection of heuristic proof techniques. The two most important ones are simplification and the invention of 'appropriate' induction arguments. The system also contains heuristics for eliminating 'undesirable' expressions, the use of equality, generalization, and the elimination of irrelevance.

Because our linear arithmetic procedure interacts with our 'simplifier' and term 'rewriter', it is necessary to explain these procedures in more detail.

To prove a formula our system first applies the simplifier to it. The *simplifier* is a procedure that takes a formula as input and returns a set of supposedly simpler formulas as output. Under the assumption that the input formula is false, it is equivalent to the conjunction of the output formulas. Since we are trying to prove the formula, it is permitted to assume its negation. If the simplifier returns the empty set of formulas we have succeeded in proving the input formula. If the simplifier returns a singleton set containing the input formula, it has failed to reduce the problem and we try some other proof technique, e.g. induction. Otherwise we try to prove, recursively, each output formula.

A formula is represented as a *clause* consisting of an implicitly disjoined set of literals. Literals are in fact *terms* of our logic. The term *p*, when used as a literal, can be thought of as the formula $p \neq F$, where $F$ is a distinguished constant in our logic.

The simplifier works by successively 'rewriting' the literals of the goal clause while assuming the complements of the remaining literals. The object is to rewrite at least one literal to $T$ (or any other non-$F$ value).

The *rewriter* is a procedure that takes a term, a substitution and a 'context' and returns a term. Among other things the context specifies a set of assumptions. The term returned by the rewriter is equal (in a certain sense determined by the context) to the result of instantiating the input term with the input substitution, under the assumptions in the context. The context contains a variety of other information which we will explain when necessary.

The rewriter applies conditional rewrite rules derived from axioms, recursive definitions, and previously proved theorems tagged 'rewrite rule' by the user. Roughly speaking, a previously proved lemma of the form:

$$h_1 \wedge h_2 \wedge \cdots \wedge h_n \rightarrow lhs = rhs$$

causes the rewriter to replace all instances of *lhs* by the corresponding instance of *rhs* provided each of the instantiated $h_i$ can be established. To establish the hypotheses the rewriter attempts recursively to rewrite them to non-$F$—a form of backwards chaining. The system contains fairly sophisticated search strategic heuristics for controlling the expansion of definitions, stopping unproductive backwards chaining, using permutative rewrites, etc.

Among the theorems proved by the theorem prover described in Boyer and Moore (1979) are: the totality, soundness, and completeness of a decision procedure for propositional calculus; the correctness of a 'toy' expression compiler; the correctness of the fastest known string-searching algorithm; and the existence and uniqueness of prime factorizations. Other proofs discovered before the linear algorithm was implemented include the correctness of a recursive descent parser (Gloess, 1980), the correctness of an arithmetic simplifier now in routine use in the system (Boyer and Moore, 1981a), and the correctness of several FORTRAN programs (Boyer and Moore, 1981b).

## 3. LINEAR ARITHMETIC

The theorem prover described above can easily prove by mathematical induction such simple theorems as:

$$X \leq Y \wedge Y \leq Z \rightarrow X \leq Z \tag{1}$$

$$X - 1 \leq X \tag{2}$$

$$0 \leq Y \rightarrow X \leq X + Y, \tag{3}$$

but because of search strategic heuristics the system cannot always employ such lemmas intelligently after they have been proved. For example, while the system would easily recognize:

$$A \leq B \wedge B \leq C \rightarrow A \leq C$$

as an instance of the transitivity of $\leq$, (1), it would not so easily prove:

$$A - 1 \leq A + B$$

where $B$ is non-negative. But $A - 1 \leq A + B$ can be derived from (2) and (3) using (1). The theorem prover described in Boyer and Moore (1979) tries to derive $A - 1 \leq A + B$ from (1)–(3) by rewriting. In particular, it observes that the result follows from (1) if one instantiates $X$ with $A - 1$ and $Z$ with $A + B$, chooses an appropriate instantiation of the intermediate variable $Y$, and backward chains to relieve the two hypotheses. If the instance chosen for $Y$ is $A$, then the two hypotheses are immediate from (2) and (3) and the assumption that $B$ is non-negative. But our system is unable to guess that an appropriate choice for $Y$ is $A$. After failing to find a proof by simplification, the system in Boyer and Moore (1979) proves the inequality by induction on $A$.

Now suppose that in a subsequent proof the rewrite routine is faced with the task of relieving the hypothesis $A - 1 \leq A + B$. Since it cannot derive this inequality from (1)–(3) by rewriting alone, and since we do not try induction to relieve hypotheses, the inequality hypothesis can be established only if $A-1 \leq A+B$ (or some mild variation of it) is available explicitly as a previously proved rewrite rule. There are two undesirable aspects to this situation. First, the search space for rewrites about $\leq$ gets very large because it contains many derived facts involving transitivity and addition. Second, the user is obliged to recognize when the system is failing to find a proof because of its lack of knowledge of such composite 'linear' facts and to state such *ad hoc* lemmas explicitly.

But the integers are probably the most important objects in the mathematics of computer programming. Facts about the integers must be second nature to any practical theorem prover for program analysis. Therefore, after having convinced ourselves of the power of our underlying heuristics, we decided to build-in a linear arithmetic procedure.

The naturals, $\mathcal{N}$, or Peano numbers are the most primitive inductively constructed domain in our theory. Many other domains are constructed on top of the naturals (e.g. the 'atomic symbols' or literal atoms, the integers, the rationals). Thus, we decided to build-in a procedure for deciding some linear inequalities over $\mathcal{N}$, i.e. universally quantified formulas involving the equality $(=)$ and Peano 'less than' $(\prec)$ relations, the natural constants $(0, 1, 2, \ldots)$, and the Peano addition $(\oplus)$ and subtraction $(\dot{-})$ functions. The syntax of our logic is actually the prefix syntax of LISP. We adopt infix here for the purposes of exposition. For readers familiar with Boyer and Moore (1979) or our theorem prover, $x \in \mathcal{N}$ here denotes (NUMBERP $x$), $x \prec y$ denotes the term (LESSP $x$ $y$), $x \preccurlyeq y$ denotes (NOT (LESSP $y$ $x$)), $x \oplus y$ denotes (PLUS $x$ $y$), and $x \dot{-} y$ denotes (DIFFERENCE $x$ $y$).

While our logic provides many different types of objects it does not have a typed syntax. Thus, $T \oplus 3$ is a well-formed term. Our definitions of $\prec$, $\preccurlyeq$, $\oplus$, and $\dot{-}$ 'coerce' non-natural arguments to 0. Thus, $T \oplus 3 = -1 \oplus 3 = 3$ and $T \prec 4$ is true but $-3 \prec 0$ is false. In addition, the Peano subtraction function returns 0 if the minuend is smaller than the subtrahend, i.e. $x \dot{-} y = 0$ if $x \prec y$, so $5 \dot{-} 8 = 0$.

In this paper we use the more familiar signs, $<$, $+$, $-$, and $*$ to denote less than, sum, difference, and multiplication, respectively, over the integers or rationals (according to context).

Linear integer arithmetic, and thus linear Peano arithmetic, is decidable. However, integer decision procedures (e.g. Cooper, 1972) are quite complicated compared to the many well-known decision procedures for linear inequalities over the rationals (King, 1969; Hodes, 1971; Bledsoe, 1975; Shostak, 1977, 1978). Therefore, following the tradition in program verification, we adopted a rational-based procedure, exploiting the observation that if a conjunction of inequalities is unsatisfiable over the rationals it is unsatisfiable over the integers. Such a procedure is sound but incomplete. For example, $2*X=3$ is satisfiable over the rationals but not over the integers. Thus, a rational procedure would not prove $2*N\neq 3$, where $N$ is an integer. Of course, it is hoped such theorems do not arise frequently in program verification (Nelson and Oppen, 1979). Should they, we might prove them by more powerful methods (e.g. induction).

For efficiency reasons, the rational method we eventually adopted was based on that described in the literature by Hodes (1971). The algorithm is just a formalization of the high school idea of 'cross-multiplying and adding' equalities to eliminate variables.

In its simplest form the algorithm is used to detect unsatisfiability in a conjunction of linear inequalities over the rationals. The first step is to convert each inequality into a 'normalized polynomial inequality' (or simply 'polynomial') by collecting like terms, cancelling when possible, and making all coefficients integers. Thus, the expression:

$$2+X-(Y+Z)\leq(A-X)-1$$

is 'linearized' to

$$3-1*Z-1*Y+2*X-1*A\leq 0.$$

Then, working one's way down through some ordering on the multiplicands, one eliminates one multiplicand at a time from the set of polynomial inequalities by cross-multiplying coefficients and adding inequations so as to cancel out the selected multiplicand in all possible ways. Eventually one obtains a set of ground inequalities whose validity may be determined by evaluation and which is satisfiable over the rationals iff the initial set is.

To apply such a procedure to problems over the integers it is convenient to adopt $\leq$ as the main connective and to transform $X<Y$ into $X+1\leq Y$. By making explicit the information that distinct integers are separated by at least 1, fewer valid integer inequalities 'fall between the cracks' in the rationals. An equality, such as $X=Y$, is handled as though it were the conjunction $X\leq Y\wedge Y\leq X$; a negative equality, such as $X\neq Y$, is handled as though it were the disjunction $X<Y\vee Y<X$ (i.e. the main conjunction must be split into two cases).

To apply the procedure to the naturals as they are axiomatized in our logic, we must take precautions to ensure that all quantities involved in arithmetic expressions are of type $\mathcal{N}$ and that the minuend of each $\dot{-}$ expression is no smaller than the subtrahend. For example, we can linearize $X\neq Y$ to $1-Y+X\leq 0\vee 1+Y-X\leq 0$ only if we know both $X\in\mathcal{N}$ and $Y\in\mathcal{N}$. Failure to consider the non-numeric case would permit the linear arithmetic procedure to prove the non-theorem $X\preccurlyeq Y\wedge Y\preccurlyeq$

$X \rightarrow X = Y$.† Similarly, it is permitted to linearize $X \leqslant Y \dot{-} Z$ to $Z - Y + X \leq 0$ only when $Z \leqslant Y$. We call such additional conditions as $X \in \mathcal{N}$ and $Z \leqslant Y$ 'linearization hypotheses.' Roughly speaking, we assume the linearization hypotheses necessary to obtain the normalized polynomials for the main conjecture. If the linear arithmetic procedure is able to prove the main conjecture under those hypotheses we set ourselves the task of proving the main conjecture under the negation of each linearization hypothesis.

Readers troubled by our desire to handle the naturals instead of the integers or by the lack of typing in our language should not be discouraged from reading on. These aspects of the problem only contribute in minor ways to the difficulty of using linear procedures.

Readers troubled by our selection of such a simple and old-fashioned decision procedure are invited to reflect upon the fact that an instantaneous oracle for deciding linear arithmetic problems like those above would increase the speed of our theorem prover on typical program verification problems by less than 3%. Furthermore, as a cursory reading of Section 8 reveals, our final linear arithmetic procedure is an implementation of the above procedure in much the same sense that the software for the Space Shuttle is an implementation of Newton's laws.

## 4. ADEQUACY OF LINEAR ARITHMETIC PROCEDURES

Before we describe how we have combined the above decision procedure with our more powerful heuristic techniques we address the question 'is linear arithmetic alone sufficient?'. Of course, one must ask 'sufficient for what?'. Since our major concern is the mechanization of the mathematics underlying computer program analysis, we focus our attention on proofs of program correctness. In this context linear arithmetic procedures, and particularly those that are decision procedures only on the rationals, are far from adequate. In this section we discuss three simple program verification exercises that involve either valid integer inequalities that are invalid over the rationals or that are non-linear.

A problem of the first type arose in the first program we tried to verify after adding a linear procedure. The specification of the program was to implement a simple table look-up scheme. The implementation of the table was an array of positive even length, $D$, with keys and their associated values stored in alternate locations. The access program searched the array linearly, pushing the current index, $I$, up from 1 in increments of 2, stopping when $I > D$. One of the verification conditions established that when the winning key is found at $I$ and the associated value is fetched from $I + 1$ no array-bound violation occurs. That is, if the current index, $I$, is a positive odd integer and $I \leq D$, then $I + 1 \leq D$. By

† A counterexample to the conjecture is obtained by letting $X$ be $T$ and $Y$ be $F$.

letting $D$ be $2 * L$ and $I$ be $2 * K + 1$, where $L$ and $K$ are arbitrary positive integers, we can cast the problem as a linear arithmetic problem:

$$0 < K \wedge 0 < L \wedge 2 * K + 1 \leq 2 * L \rightarrow 2 * K + 2 \leq 2 * L.$$

This is a theorem over the naturals. However, it is not a theorem over the rationals and so does not yield to a rational-based linear arithmetic procedure. Our modified system proved it by induction.

More frequently we see arithmetic problems that do not fall into the linear domain at all. The classic verification example, Euclid's gcd algorithm, illustrates this. Consider the *verification condition* (*v.c.*) that states that if $X < Y$ then the largest number that divides $X$ and $Y$ is also the largest number that divides $X$ and $Y - X$. Some attempts to verify Euclid's algorithm assume this v.c. We wish to prove it. The key step in the proof is that if $X < Y$ and $Z$ divides $X$ then $Z$ divides $Y$ iff $Z$ divides $Y - X$. The definition of '$I$ divides $J$' is that ($J$ mod $I$) = 0, where mod is defined recursively. This problem falls outside linear arithmetic.

A much more mundane example arises in the attempt to implement a two-dimensional array access module on top of a linear storage scheme. The element at position $I,J$ in the two-dimensional array is mapped to location $I + D * J$ in the linear array. One of the v.c.s establishes that every pair of distinct points in the two-dimensional array maps to a pair of distinct points in the one-dimensional array. Roughly speaking one wishes to establish that $I_1 + D * J_1 = I_2 + D * J_2$ iff $I_1 = I_2$ and $J_1 = J_2$. Because of the multiplication by $D$, this problem falls outside of linear arithmetic. One might assume this obvious fact. However, attempts to prove it—by induction (on $J_1$ and $J_2$ simultaneously) or by appeals to inductively proved facts about mod and quotient—reveal that as stated it is not a theorem. One must hypothesize that $I_1$ and $I_2$ are legal indices in the two-dimensional array, i.e. that $I_1 < D$ and $I_2 < D$.

Our point is not to say that linear arithmetic is useless. We have invested several years in building it into our system and have seen it help out in the verification of very many programs. Our point is that it is not unusual to see programs—mundane, everyday programs—that require the proof of arithmetic theorems beyond those of linear arithmetic. If a verification system cannot establish such results then one is forced to assume, rather than prove, many verification conditions.

## 5. SIMPLE INTEGRATION STRATEGIES

In this and the next section we sketch the evolution of our linear arithmetic procedure and its use by the theorem prover described in Boyer and Moore (1979). We then illustrate the procedure in use. Our intention in these sections is to motivate some of the elaborate bells and whistles described in Section 8.

Our goal was simple: build in enough information about the naturals so that is no longer necessary for the rewriter to consider explicitly those rewrite rules expressing the truths of linear arithmetic. For example, if some old proof required an explicit appeal to the transitivity of $<$ and some inspired 'guess' instantiating the intermediate variable, then after building in linear arithmetic we should be able to find that proof without an explicit appeal to transitivity or a heuristic guess. Ideally, achieving this goal should speed up the theorem prover because certain facts are built-in and because the search space of lemmas is reduced by the deletion of many derived truths of linear arithmetic. Furthermore, achieving the goal frees the user from having to bring linear facts to the system's attention by proving them as rewrite rules.

Our first attempt at incorporating a linear arithmetic procedure was to add it as a 'black-box' applied to every formula produced by the simplifier. The procedure took as its input a clause to prove. If the conjunction of the negations of the literals in the clause is unsatisfiable, the clause is valid. The linear procedure extracted the inequalities in the clause, negated them, introduced linearization hypotheses, formed the set of normalized polynomials and tested the set for unsatisfiability as described. If the set was not found unsatisfiable, the theorem prover tried the next proof technique in its repertoire. If the set was found unsatisfiable, the linear procedure produced as its output a set of clauses, each obtained by adding the negation of a linearization hypothesis to the input clause. The theorem prover then recursively set out to prove each of those 'pathological' cases.

When we applied the modified theorem prover to the 404 definitions and theorems in Appendix A of Boyer and Moore (1979) the linear arithmetic procedure contributed to almost no proof except those of rewrite rules expressing linear facts. That is, when the theorem prover was working on interesting theorems, the simplifier did not produce many conjectures that yielded to linear arithmetic.

Here is an example. One of the uninteresting rewrite rules proved by the linear procedure was named GT.SUB1 and stated that $X \dot{-} 1 \leqslant X \oplus Y$. The only reason this rewrite rule was in the list was because it was needed in the proof of an interesting verification condition later in the list, named FSTRPOS.VC7. The linear procedure in the modified theorem prover established GT.SUB1 immediately but the linear procedure did not participate in the proof of FSTRPOS.VC7. In fact, the proof of FSTRPOS.VC7 still required the explicit use of GT.SUB1 as a lemma and if that lemma was absent then the proof attempt failed—even though the lemma was built-in.

The problem was that while GT.SUB1 was built into the modified theorem prover it was not built-in in the right place. The proof of FSTRPOS.VC7 used GT.SUB1 to relieve a hypothesis of another lemma, not to prove a simplified part of the main theorem. In our experience linear arithmetic reasoning is most often required during term rewriting and is not terribly useful if its only role is to establish simplified v.c.s.

How can we move the linear arithmetic procedure into the rewriter? Recall that the rewriter operates in a context of assumptions. Suppose we wish to establish an inequality—say a hypothesis of a rewrite rule we wish to apply. We may do so as follows: negate the inequality, conjoin it to the inequalities among our assumptions, linearize the inequalities to obtain a set of polynomials, and then apply the cross-multiply and add procedure to detect unsatisfiability. If the set is found unsatisfiable, the inequality is valid under our assumptions. (Since assumptions of the form $x \neq y$ generate disjunctions of polynomials we will for the time being simply discard any such assumptions.)

For efficiency, we implement this test incrementally. We store the assumptions in a pre-processed internal form in which all polynomials have been maximally 'propagated' in the sense that every admissible cross-multiply and add has been performed. A cross-multiply and add is admissible under our propagation rules only if it eliminates the 'heaviest' multiplicand in both inequalities. For example, because $F(G(X))$ has greater weight than $G(X)$, a cross-multiply and add involving the polynomial

$$8+F(G(X))-G(X)\leq 0$$

is permissible only if it eliminates $F(G(X))$. Thus, if no inequality in the incremental data base has $F(G(X))$ occurring negatively as the heaviest multiplicand we do not propagate the above polynomial even if there are other polynomials about $G(X)$. Such propagation will occur as soon as $F(G(X))$ is eliminated and $G(X)$ is exposed as the heaviest multiplicand. This reduces the amount of work the procedure does if irrelevant polynomials are present.

The 'heavier' relation is a total ordering on terms. We say $t_1$ is *heavier* than $t_2$ iff either the number of variables in $t_1$ is greater than that in $t_2$, or the number of variables in the two are equal but the 'size' of $t_1$ is greater than that of $t_2$, or the number of variables in and the sizes of the two are equal and $t_1$ comes later than $t_2$ in the lexicographic ordering of terms. By *size* we mean the number of open parentheses in the unabbreviated presentation of the term.

When we introduce a new polynomial into a data base and perform all admissible cross-multiplies and adds, we say we have 'pushed' the polynomial into the data base. The result is a new data base representing the conjunction of the old assumptions and the new inequality.

If pushing a polynomial destructively modifies the initial data base one needs a 'pop' or 'undo' operation; otherwise the attempt to establish a hypothesis by assuming its negation would permanently alter our assumptions. If pushing an inequality does not destructively modify the data base, the initial data base may be recovered by the usual variable binding mechanisms. This aspect of the problem detracts from the efficiency and simplicity of linear algorithms that rely upon destructively modified cyclic structures (e.g. Shostak, 1978) since the 'pop' algorithm is usually messy. The simple procedure we chose allows the data base to be an ordered alist (associating polynomials with the heaviest multiplicand in them) and permits the implementation of a non-destructive push operation that constructs the new data base from the old using little new structure.

Using such a scheme we programmed the rewriter to use the linear arithmetic procedure when trying to establish inequalities. We found that while the new system was an improvement over the earlier one, our goal—of eliminating the need for explicit lemmas expressing linear facts—was far from achieved.

The problem arises from the presence of interpreted functions. Here is a simple, artificially constructed example. Suppose one needs to prove:

$$L \leqslant \mathrm{MIN}(A) \wedge 0 < K \rightarrow L < \mathrm{MAX}(A) \oplus K \tag{4}$$

where MIN($A$) and MAX($A$) are defined as the minimum and maximum elements of $A$. This theorem is not a consequence of linear arithmetic; in particular, since MIN and MAX are treated as uninterpreted function symbols (4) is treated as though it were:

$$L \leqslant \mathrm{MIN} \wedge 0 < K \rightarrow L < \mathrm{MAX} \oplus K.$$

However, if one adds to (4) the additional hypothesis that

$$\mathrm{MIN}(A) \leqslant \mathrm{MAX}(A) \tag{5}$$

the resulting linear arithmetic problem is equivalent to

$$L \leqslant \mathrm{MIN} \wedge 0 < K \wedge \mathrm{MIN} \leqslant \mathrm{MAX} \rightarrow L < \mathrm{MAX} \oplus K,$$

which is a theorem.

To use a linear arithmetic procedure to prove formulas like (4) it is necessary to identify 'interesting' additional hypotheses like (5) to connect multiplicands in the linearization of the goal.

Many readers may object that we should not be trying to use linear arithmetic to prove formulas like (4). But what is the alternative? If we do not use our built-in linear arithmetic procedure we are forced to derive (4) from (5) and explicit linear facts such as the transitivity of $\leqslant$. But if linear arithmetic procedures are to be useful to larger systems they should free the larger system from having to consider the truths of linear arithmetic (such as transitivity). If we do not extend our handling of linear arithmetic to take into account lemmas about 'non-linear' function symbols then the only way we will prove many arithmetic facts is to ignore the work on linear procedures altogether and return to the heuristic instantiation and chaining methods rejected earlier.

How shall we take into account facts about defined functions? We decided that if after a polynomial has been pushed into the data base no contradiction was found we would look at the multiplicands in the data base and try to link them via additional inequalities obtained by instantiating previously proved lemmas. We call this 'augmenting' the data base. For example, if we have previously proved that:

$$\text{MIN}(S) \leqslant \text{MAX}(S)$$

and construct a linear data base containing the multiplicand $\text{MIN}(A)$ or $\text{MAX}(A)$ we might push the polynomial obtained from $\text{MIN}(A) \leqslant \text{MAX}(A)$.

Of course, as Herbrand knew, the problem of which instances of which lemmas to consider is the heart of the theorem-proving problem. We therefore implemented heuristics to control the instantiation of previously proved inequalities and their addition to the polynomial data base. For example, a lemma such as $X < F(X)$ is a 'pump' that may cause one to push, successively, $N < F(N)$, $F(N) < F(F(N))$, etc. Just as with backwards chaining, one has to decide when to stop trying to add new multiplicands to the data base. Our heuristic is to use the same criteria we use to limit backwards chaining, namely, add no multiplicand that is 'worse than' every multiplicand in the data base. For example, we might go around the above loop five times if $F^5(N)$ was initially a multiplicand in the data base.

The problem is further complicated by the need to consider inequality lemmas with hypotheses. For example, let $\text{MEMB}(X, S)$ be the predicate that $X$ occurs in the sequence, $S$, $\text{LEN}(S)$ be the length of $S$, and $\text{DEL}(X, S)$ be the result of deleting all occurrences of $X$ from $S$. Then the following lemma links the theory of lists to arithmetic.

$$\text{MEMB}(X, S) \rightarrow \text{LEN}(\text{DEL}(X, S)) < \text{LEN}(S). \tag{6}$$

Suppose we are asked to prove

$$\text{MEMB}(Z, A) \wedge W \oplus \text{LEN}(A) \leqslant K \rightarrow W \oplus \text{LEN}(\text{DEL}(Z, A)) < K \oplus V.$$

The theorem is a consequence of linear arithmetic if we first add the additional information that:

$$\text{LEN}(\text{DEL}(Z, A)) < \text{LEN}(A).$$

To obtain this inequality we must first instantiate (6), replacing $X$ by $Z$ and $S$ by $A$ [so as to obtain a new inequality about the multiplicand $\text{LEN}(\text{DEL}(Z, A))$], and then relieve the hypothesis $\text{MEMB}(Z, A)$. Note that to relieve the hypothesis we may have to engage in non-arithmetic reasoning. Therefore, we relieve the hypotheses of 'linear rules' like (6) by the same methods we relieve the hypotheses of conditional rewrite rules: we recursively rewrite them under our current assumption.

As a consequence, the rewrite mechanism and the linear arithmetic procedure are mutually recursive. The rewrite mechanism calls the linear arithmetic procedure to establish certain inequalities and the linear arithmetic procedure calls the rewrite routine to establish the hypotheses of lemmas providing additional information about the multiplicands in the problem.

## 6. FURTHER REFINEMENTS

Thus far it has not been crucial to this discussion that we adopted the simple propagation procedure based on Hodes's algorithm. Indeed, at this stage in our actual experimentation we had coded several different linear arithmetic decision procedures and used them as 'black boxes'. However, the attempt to implement the use of 'linear rules' required opening up the black box. In addition, other problems, not yet discussed in detail, required significant modifications to the procedure itself.

One obvious problem is that the heuristic component of the theorem prover must be able to determine what the multiplicands in the current data base are. Either the linear arithmetic procedure should construct the set of multiplicands and make that available outside, or the heuristic component should know the structure of the internal data base. We chose the latter because it was most efficient. However, this choice blurs the line between the heuristic component and the linear procedure.

A second problem arises from the restrictions on the order in which inequalities are processed by the propagation procedure. Consider our procedure. It eliminates the heaviest multiplicands first. Thus, it is a waste of time for the heuristic component to obtain an inequality about $G(X)$ in response to a polynomial such as $8+F(G(X))-G(X)\leq 0$.

A related problem is the organization of previously proved inequalities so that the system can rapidly determine relevant facts about the key multiplicands in the data base. Suppose we want to pre-process and store a lemma whose conclusion is an inequality. We store such a lemma so that it may be accessed according to the function symbols of the terms that might, when the lemma is instantiated, become the heaviest multiplicands in the linearized form of the concluding inequality. For example, if a lemma concludes with $F(X)\oplus G(Y)\preccurlyeq H(G(Y))\oplus X$, we store it under the function symbols $F$ and $H$. We further require that each such 'key multiplicand' contain enough of the variables in the lemma so that if the key multiplicand is instantiated and the hypotheses are relieved (possibly requiring the instantiation of additional variables) the concluding inequality is fully instantiated.

Note that the notion that the linear procedure is a black box has been destroyed. Once a particular linear procedure has been selected by the implementor, an extremely large amount of work must be done to interface to it efficiently. In our case, the time taken to program the linear procedure was insignificant (one man-day) compared to the time taken to interface to it (several man-months, not counting the several man-months devoted to the empirical evaluation of each successive implementation). It is certainly not possible to substitute one linear procedure for another. But the worst is yet to come. Much to our dismay we were eventually forced to modify both the linearization subroutine and the propagation subroutine to complete the integration. Thus, the notion that we could choose a linear procedure 'off the shelf' is also destroyed.

It has been found useful by those who write verification systems for the theorem prover to report which lemmas were used in a proof. Such information is necessary if the verification system is to permit the user to redefine or re-axiomatize concepts without having to rederive the proofs of logically independent results. How can the heuristic theorem prover determine whether a given linear rule was used? A 'shotgun' approach can be used. That is, when linear arithmetic participates in a proof it can report that it used every linear rule from which a polynomial was generated and pushed. But the shotgun approach tends to make proofs depend upon many irrelevant lemmas. The approach we finally took was to modify the linearization and propagation subroutines so that every polynomial in the data base carries with it a record of the linear rules from which it was derived. This information is propagated in the obvious way as new inequalities are formed from old ones. When a contradiction is found it is possible to announce exactly which linear rules were used.

More seriously, the search for 'interesting' lemmas and the work involved in relieving their hypotheses make it more expensive to set up the data base for a clause initially. Our first approach was as follows. To rewrite a literal in a clause we pushed into an initially empty data base the polynomials derived from the negations of the remaining inequalities in the clause initially. The data base was then closed under the operation of pushing polynomials derived from heuristically chosen instances of linear rules after relieving their hypotheses. An arbitrary amount of work might be done in setting up the data base for the rewriting of a single literal. Furthermore, the work done to set up the data base for one literal is often very similar to that done for the adjacent literal. Thus, if there is a set of 'expensive' literals in the clause the work they trigger is duplicated each time an 'inexpensive' literal is set up. We found this prohibitively expensive and abandoned the idea of setting up a different data base for each literal.

While trying to prove $p$ it is permitted to assume the negation of $p$. Thus, we set up just one data base containing the negations of all the inequalities in the clause and closed under linear rules. That data base was used during the rewriting of each literal.

Of course, while working on the literal $p$ one must be very careful to avoid using the assumption that $p$ is false to rewrite $p$ to $F$ (This phenomenon happens often to students learning proof by contradiction. They assume the negation of what they wish to prove, engage in a long sequence of steps, and then announce that the 'theorem is false'.) 'Accidentally' replacing a literal by $F$ is sound but risky: the rewritten literal is dropped from the disjunction being proved and one is forced to prove a stronger goal which may in fact be invalid and hence unprovable. When this occurs we say the simplifier has 'bitten its own tail'. If the literal being rewritten is assumed false in the context of the rewriter then special precautions must be taken.

Our first attempt to keep the simplifier from biting its own tail was to prevent any attempt to push into the data base the literal we are trying to simplify. This method failed to be effective because the inequality being pushed might be different but linearly equivalent to the one being simplified.

The presence of conditional rewrite rules also complicates the situation. For example, suppose the system knows the lemma:

$$X < Y \rightarrow (X \oplus 1 < Y) \leftrightarrow (X \oplus 1 \neq Y). \tag{7}$$

This permits $(X \oplus 1 < Y)$ to be rewritten to $(X \oplus 1 \neq Y)$ under the condition that $X < Y$. Suppose the current literal is $\neg(A \oplus 1 < B)$. We assume its complement, $A \oplus 1 < B$, and begin simplifying $\neg(A \oplus 1 < B)$. The rewriter observes that it can use (7) to simplify $(A \oplus 1 < B)$ to $(A \oplus 1 \neq B)$ if it can establish $A < B$. By appealing to linear arithmetic it derives $A < B$ from $A \oplus 1 < B$. Therefore, it simplifies $\neg(A \oplus 1 < B)$ to $A \oplus 1 = B$, biting its own tail. We abandoned the hope that we could easily avoid tail biting when we saw such examples. (Less pathological examples can be constructed if one considers lemmas about user-defined function symbols.)

The solution we finally adopted required the further elaboration of the linear algorithm itself. We programmed the linearization subroutine to attach to each polynomial the set of literals from which it was derived. In most cases this is a singleton set containing the inequality literal itself, but in some cases (as when the linearization depends upon another literal to relieve linearization hypotheses) it contains multiple literals. We programmed the propagation subroutine to merge these sets as new polynomials are formed. Thus, we know which literals are involved in the derivation of each polynomial in the data base. Finally, we programmed the propagation subroutine to avoid using any polynomial whose derivation involves the literal we are currently trying to rewrite or any literal previously rewritten to $F$. (The latter restriction prevents another form of tail biting. Suppose $p$ and $p'$ are two equivalent but non-identical inequality literals. Consider simplifying the clause $\{p p'\}$. The polynomial data base contains two equivalent polynomials one descending from $\neg p$ and the other from $\neg p'$. While rewriting $p$, we use the polynomial descending from $\neg p'$ to derive $p = F$. If we permit ourselves, while rewriting $p'$, to use the old polynomial descending from $\neg p$ we will bite our tail.)

Another problem we faced is dealing with the linearization hypotheses. These hypotheses are generated when inequalities are put into polynomial form and must be relieved (i.e. either by proving them from other hypotheses or by splitting on them and proving the entire conjecture assuming each of them false).

For example, if the linear arithmetic procedure is applied to:

$$W < I \dot{-} K \wedge J < K \rightarrow W \oplus J < I$$

then, under the hypothesis $K \leqslant I$ the formula is found valid. But an additional case, obtained by assuming that $I < K$ and $I \dot{-} K = 0$, must be proved:

$$I < K \wedge W < 0 \wedge J < K \rightarrow W \oplus J < I.$$

Our first attempt to handle this problem took the shotgun approach again. That is, if the linear arithmetic procedure participated in the simplification of a clause we added to the simplified clause(s) the linearization hypotheses for each literal in the input clause. Furthermore, we produced as new goals additional versions of the input clause in which each of these added hypotheses is negated. Once again, the advantage to the shotgun approach is that while we had to modify the subroutine for putting literals into normal form we did not have to modify the propagation subroutine itself.

But the difficulty with the shotgun approach was that it caused many irrelevant splits. For example, suppose the theorem to be proved is:

$$X < Y \wedge J \leqslant I \dot{-} K \rightarrow X < Y \oplus 1.$$

The contradiction found is actually derived from the first hypothesis and the conclusion. The second hypothesis is irrelevant. But the $I \dot{-} K$ expression in it gives rise to the additional case:

$$X \leqslant Y \wedge I < K \wedge J \leqslant 0 \rightarrow X \leqslant Y \oplus 1.$$

This case will be proved exactly as before, but it need not have arisen in the first place and, in general, reproducing the proof may be quite expensive because of the need for lemmas. Furthermore, the number of irrelevant splits grows exponentially with the number of irrelevant difference or predecessor expressions in the clause. Unfortunately, irrelevant hypotheses are common in mechanically generated formulas.

For example, in our system's first proof of the termination of the Takeuchi function (Moore, 1979) the proof of one lemma involved 412 cases, many of which were irrelevant.

One solution to this problems is that adopted for the tail biting problem. If the linear procedure keeps track of which literals are involved in the derivation of each polynomial, it is possible to report which literals are involved in the eventual contradiction found. Then one can split on the hypotheses necessary to obtain the polynomial for those literals.

However, one can do better. Recall that an assumption such as $I \dot{-} J = K$ gives rise to two inequalities, $I \dot{-} J \leqslant K$ and $K \leqslant I \dot{-} J$. If one is asked to assume $I \dot{-} J \leqslant K$ it is permitted to assume $-K - J + I \leq 0$ even if $I \leqslant J$. That is, the polynomial for the first inequality can be obtained without any additional hypothesis. However, if one is asked to assume $K \leqslant I \dot{-} J$, the polynomial, $K + J - I \leq 0$, may be obtained only under the additional hypothesis that $J \leqslant I$. It is possible that only one of the two polynomials will participate in a contradiction. Thus, in our implementation of the linearization subroutine we attach to each polynomial its linearization hypotheses, we propagate that information in the obvious way, and split on the hypotheses necessary for those polynomials used in the eventual contradiction found. This eliminates many irrelevant case splits when dealing with the naturals. For example, in the proof of the above-mentioned lemma about Takeuchi's function, 311 of the 412 cases were eliminated.

There is one remaining aspect of our scheme. If is often the case that the linear arithmetic procedure derives two polynomials $-y + x \leq 0$ and $y - x \leq 0$. That is, under the hypothesis that $x$ and $y$ are both naturals, they are equal. While the knowledge of this equality is available to the linear arithmetic procedure it is not known to the rewriter. Therefore, after we have set up the polynomial data base for a clause but before we begin rewriting the literals of the clause, we search the data base for 'mated' pairs of polynomials as above and under certain circumstances add equality hypotheses to the clause.

This concludes the casual description of how we integrated a linear arithmetic procedure into our heuristic theorem prover. The objective of these two sections has been to substantiate our assertion that integrating such a procedure into a larger system is quite difficult and frequently requires discarding the notion that the procedure is a black box. In addition, we have attempted to motivate the rather elaborate data structures and procedures described in Section 8.

## 7. TWO EXAMPLES

In this section we present two examples illustrating the co-operation between our heuristic theorem prover and its linear arithmetic procedure.

The first example comes from our system's proof of the correctness of the Boyer–Moore fast string-searching algorithm (Boyer and Moore, 1977). The algorithm searches for the first occurrence of a given pattern in a given text. Both the pattern and text are strings of characters over a finite alphabet. The algorithm uses an array that associates with each character in the alphabet the distance between the last occurrence of that character in the pattern and the end of the pattern. For those characters in the alphabet that do not occur in the pattern the array contains the length of the pattern. A previously verified subroutine initializes this array. In particular, after the call of this subroutine the $C$th element of the array is known to be DELTA1$(PAT, LP, C)$, where DELTA1 is defined recursively to be the distance specified above, $PAT$ is the input pattern and $LP$ is the length of $PAT$. DELTA1 is another example of an interpreted function symbol needed to specify a program. An interesting inequality involving DELTA1 is DELTA1.LESSEQP.PATLEN, which states that DELTA1$(PAT, LP, C) \leqslant LP$. This result can be proved by induction on $LP$.

Repeatedly during the execution of the string-searching algorithm the current index into the text, $I$, is incremented by the $C$th element of the above array. One must prove that this addition does not cause an arithmetic overflow. The input assertion of the program assures us that the sum of the length of $PAT$, $LP$, and that of the text, $LT$, is less than or equal to the maximum representable positive integer, MAXINT. Furthermore, at the time of the increment we know that $I$ is less than or equal to $LT$. We must prove:

$$LP \oplus LT \leqslant \text{MAXINT} \wedge I \leqslant LT$$

$$\rightarrow$$

$$I \oplus \text{DELTA1}(PAT, LP, C) \leqslant \text{MAXINT}.$$

This is not a consequence of linear arithmetic alone. However, after pushing the above inequalities into the data base, DELTA1$(PAT, LP, C)$ is a heaviest multiplicand in an inequality in the data base. By appealing to DELTA1.LESSEQP.PATLEN we obtain the additional information that DELTA1$(PAT, LP, C) \leqslant LP$, from which the above conjecture follows by linear arithmetic.

A second example comes from our program's proof that a certain tree-normalization algorithm terminates. The proof involves showing that the measure MS of the nested ordered pair $\langle\langle a, b\rangle, c\rangle$ is strictly greater than MS of $\langle a, \langle b, c\rangle\rangle$, where MS is defined by the user as follows:

$$\text{MS}(\text{atm}) = 1, \text{ if atm is not a pair, and}$$

$$\text{MS}(\langle x, y\rangle) = \text{MS}(x) \times \text{MS}(x) \oplus \text{MS}(y)$$

where $\times$ is the Peano multiplication function.

Note that by induction one can prove $0 < \textsc{ms}(X)$.

Consider our goal:

$$\textsc{ms}(\langle a, \langle b, c\rangle\rangle) < \textsc{ms}(\langle\langle a, b\rangle, c\rangle). \tag{8}$$

After simplifying by expanding the definition of MS several times and applying such previously proved arithmetic rewrite rules as the associativity and commutativity of addition and the distributivity of multiplication over addition the goal becomes:

$$\textsc{ms}(c) \oplus \textsc{ms}(a)^2 \oplus \textsc{ms}(b)^2 < \textsc{ms}(c) \oplus \textsc{ms}(b)^2 \oplus 2 \times \textsc{ms}(a)^2 \times \textsc{ms}(b) \oplus \textsc{ms}(a)^4, \tag{9}$$

where $\textsc{ms}(x)^2$ is an abbreviation for $\textsc{ms}(x) \times \textsc{ms}(x)$ and $\textsc{ms}(x)^{n+1}$, for $n > 1$, is an abbreviation for $\textsc{ms}(x) \times \textsc{ms}(x)^n$.

Upon trying to simplify the above inequality we push into an empty data base the polynomial obtained from its negation:

$$\textsc{ms}(a)^4 + 2 * \textsc{ms}(a)^2 \times \textsc{ms}(b) - \textsc{ms}(a)^2 \leq 0$$

from which we hope to derive a contradiction. No linear contradiction is found. Therefore we note that $\textsc{ms}(a)^4$ is the heaviest multiplicand and search for linear rules about $\times$. We find the linear rule

$$0 < I \rightarrow J \leqslant I \times J$$

and instantiate it by replacing $I$ with $\textsc{ms}(a)$ and $J$ with $\textsc{ms}(a)^3$ to produce:

$$0 < \textsc{ms}(a) \rightarrow \textsc{ms}(a)^3 \leqslant \textsc{ms}(a)^4.$$

The hypothesis is rewritten to $T$ by appealing to $0 < \textsc{ms}(X)$. We then heuristically decide whether we wish to push the polynomial produced from the concluding inequality. Even though it contains $\textsc{ms}(a)^3$, which is a new multiplicand, we decide it is no worse than the existing $\textsc{ms}(a)^4$ [indeed, it is a subterm of $\textsc{ms}(a)^4$]. By pushing the concluding polynomial into the data base and cancelling it against the negated goal we obtain:

$$2 * \textsc{ms}(a)^2 \times \textsc{ms}(b) + \textsc{ms}(a)^3 - \textsc{ms}(a)^2 \leq 0.$$

No contradiction is found so we again look for linear rules about the heaviest multiplicands. $\textsc{ms}(a)^2 \times \textsc{ms}(b)$ is the heaviest multiplicand in the new polynomial [since it is the same size as $\textsc{ms}(a)^3$ but $b$ is lexicographically larger than $a$]. We appeal to the same lemma about $\times$, relieve the hypothesis in exactly the same way as before, once again approve the new conclusion as being no worse than existing polynomials and push:

$$-\textsc{ms}(a)^2 \times \textsc{ms}(b) + \textsc{ms}(a) \times \textsc{ms}(b) \leq 0.$$

Propagation produces the new polynomial:

$$\textsc{ms}(a)^3 + 2 * \textsc{ms}(a) \times \textsc{ms}(b) - \textsc{ms}(a)^2 \leq 0.$$

Again no linear contradiction is found. This time the heaviest term is $\text{MS}(a)^3$. Appealing again to our lemma about multiplication we obtain and push the polynomial:

$$-\text{MS}(a)^3 + \text{MS}(a)^2 \leq 0.$$

Cancelling again produces:

$$2 * \text{MS}(a) \times \text{MS}(b) \leq 0.$$

No contradiction is found. This time the largest multiplicand is $\text{MS}(a) \times \text{MS}(b)$. Appealing again to our $\times$ lemma we obtain and push:

$$-\text{MS}(a) \times \text{MS}(b) + \text{MS}(b) \leq 0$$

which produces

$$2 * \text{MS}(b) \leq 0.$$

Again no linear contradiction is found. But this time $\text{MS}(b)$ is the heaviest multiplicand. We search for lemmas about MS and obtain:

$$0 < \text{MS}(b)$$

which, when linearized is

$$1 - \text{MS}(b) \leq 0.$$

Pushing this polynomial produces $2 \leq 0$, which is a contradiction. Thus the goal has been proved.

It takes our system a total of 22.3 seconds to prove the goal, equation (8). We can break the proof down into two phases: producing from (8) the fourth-degree polynomial (9), and appealing to linear arithmetic reasoning to prove (9). The first phase, which consists of expanding the definition of MS and applying previous proved rewrite rules, takes a total of 16.5 seconds. However, five of those seconds are consumed by attempts to prove the theorem by linear arithmetic before the normalized polynomial (9) is produced. The second phase—in which the linear arithmetic interface performs the iterated sequence of pushes and lemma instantiations leading to the final contradiction—consumes 5.8 seconds. The times measured are DEC KL-10 c.p.u. seconds consumed while running compiled INTERLISP (not counting garbage collection times and the time taken to output the proof). During the four year period this research was conducted we converted our system from INTERLISP to the MACLISP family. The MACLISP version of the system runs about twice as fast as the INTERLISP version. However, all experimental statistics in this paper are based on the INTERLISP version.

## 8. THE CURRENT IMPLEMENTATION

In this section we describe precisely the current (if not the final) version of the linear arithmetic procedure.

### 8.1. More background on the rewriter

Since the linear arithmetic procedure is mutually recursive with the new rewriter, the description of the two are intertwined. We here explain in greater detail aspects of the new rewriter that are mentioned as we describe the linear arithmetic procedure.

The rewriter takes a term, a substitution, and a context and returns a term, a set of linearization hypotheses, and the list of all rewrite and linear rules used to derive the result. The context specifies, among other things, some assumptions and the sense of equality to be maintained by the rewriter. The primary specification satisfied by the rewriter is that under the assumptions in the context plus the returned linearization hypotheses, the output term is equal (in the specified sense) to the instantiation of the input term with the input substitution. Two senses of equality are supported: propositional equivalence and identity. Two terms, $p$ and $q$, are *propositionally equivalent* iff either $p = F$ and $q = F$ or $p \neq F$ and $q \neq F$. Thus, 3 and $T$ are propositionally equivalent. When rewriting the literals of a clause, the hypotheses of lemmas, and the tests of IFs it is sufficient to maintain propositional equivalence only. At all other times we maintain identity.

Fundamental to the rewriter and to linearization is the notion of 'type sets' and 'type alists'.

A *type alist* is an association list pairing terms with 'type sets'. If $r$ is a 'shell recognizer' then we denote by $\boldsymbol{r}$ the set of all $x$ such that $(r\,x) = T$; we call $\boldsymbol{r}$ a *type*. For example, **TRUEP** is the set $\{T\}$, **FALSEP** is the set $\{F\}$, **NUMBERP** is the set of all natural numbers, and **LISTP** is the set of all ordered pairs constructed with CONS. In addition, we define one additional type containing all the non-shell objects. A *type set* is a set of types. If the term $t$ is associated with the type set $\{\boldsymbol{r}_1 \cdots \boldsymbol{r}_n\}$ on the type alist then we are assuming that $t$ is an element of one of the $\boldsymbol{r}_i$'s.

Type alists are used in the rewriter to record the assumptions governing the term being rewritten. This mechanism is discussed at length in Chap. V of Boyer and Moore (1979). When the simplifier applies the rewriter to a literal it supplies as part of the context a type alist encoding the assumptions that all the other literals of the clause are false.

In addition to a type alist, the context contains a polynomial data base, set up by the simplifier by pushing the negations of all the inequalities of the clause. The dependence on literals of each polynomial in this data base is carefully tracked so that the rewriter can ignore polynomials descending from the current literal and literals previously rewritten to $F$. Because of the ubiquity of type sets the rewriter does not record which type set assumptions were used by a rewrite. This makes it difficult to track the dependencies of polynomials derived from linear rules when setting up the initial data base and we will describe several kludges to mitigate these difficulties.

The polynomial data base is used by the rewriter when it encounters an inequality: if the negation of the inequality, when pushed into the data base, produces a contradictory polynomial, we rewrite the inequality to *T*. But we must note the hypotheses governing the contradictory polynomial and report them with our final answer if the reduction of this inequality to *T* is part of the derivation of that answer. It may not be: the inequality just established might be the first of two hypotheses of a rewrite rule. If we fail to establish the second one the work done on the first is irrelevant.

To keep track of the hypotheses generated by a rewrite, we use a push down stack called the *hyps stack* consisting of *frames* each of which contains a set of hypotheses. It is assumed that when the rewriter is called a frame has been pushed onto the hyps stack. The rewriter is implemented so that it adds to the top frame of the stack all of the hypotheses assumed during the derivation of the answer delivered. For example, suppose we are rewriting some term *t* and try to apply a rewrite rule with hypotheses. Then we push an empty frame on the hyps stack and rewrite the hypotheses of the rule, accumulating the linearization hypotheses in the new frame. If all the hypotheses of the rule are relieved and our heuristics permit us to apply the rule, we pop the new hyps frame off the stack and union its contents into the frame below (which is accumulating the linearization hypotheses actually used to rewrite *t*). If, on the other hand, some hypothesis of the rule is not relieved, we pop the new frame off the stack and throw its contents away.

A similar stack, called the *lemma stack* is used to keep track of the axiom, definition, and lemma names used by a given rewrite.

The context in which a rewrite takes place thus contains:

TA: a type alist encoding the assumption that each literal of the current clause is false (except the literal we are in the process of rewriting) and, during recursive calls to the rewriter from within the rewriter, the assumptions of the truth or falsity (as appropriate) of the tests of IF's governing the occurrence of the term being rewritten.

DB: a polynomial data base encoding the assumption that every arithmetic literal in the clause is false.

LITS-TO-BE-IGNORED-BY-LINEAR: a list containing the current literal and all previous literals of the goal clause rewritten to *F*. Polynomials in DB that descend from any literal on this list are ignored by the propagation routine.

LITS-THAT-MAY-BE-ASSUMED-FALSE: the clause being simplified, during the initial construction of the linear arithmetic data base.

HEURISTIC-TA: a type alist used for heuristic purposes when setting up the initial polynomial data base.

OBJECTIVE: a flag that tells the rewriter whether it should try to rewrite the input term to *T*, to *F*, or to anything it can. If the rewriter is 'trying' to get to *T* it does not attempt to apply rewrite rules that would replace the term by *F*. The flag is used to direct the rewriter's efforts when trying to establish the hypotheses of rewrite and linear rules.

ID/IFF: a flag specifying whether identity or propositional equivalence is to be maintained.

hyps stack: a stack of frames containing linearization hypotheses. When the rewriter returns, the assumed hypotheses will have been unioned into the top-most frame.

lemma stack: a stack of frames containing lemma names (and literals from LITS-THAT-MAY-BE-ASSUMED-FALSE used to relieve hypotheses). When the rewriter returns, the lemmas and literals used will have been unioned into the top-most frame.

history: a record of the ancestry of the current clause and what proof techniques were involved in producing each clause in the ancestry.

Other aspects to the context, not relevant to the current discussion, include such search strategic information as the stack of lemmas through which we are currently backwards chaining and a flag indicating whether the term being rewritten is textually within the clause or is part of a lemma or definition.

In addition, of course, the context implicitly contains a set of rewrite rules and linear rules derived from axioms, definitions, and previously proved theorems. This set of rules is here called the *library*.

This elaborate notion of 'context' is used implicitly not only by the rewriter but also by the linearization and augmentation procedures.

## 8.2. Polynomials

A *polynomial* is a five-tuple, $\langle c, alist, hyps, lits, supports \rangle$. The first field, *c*, is an integer constant. The second, *alist*, is a list of pairs $\langle t_i, k_i \rangle$, where each $t_i$ is a term, called a *multiplicand*, and each $k_i$ is an integer, called the *coefficient* of the corresponding multiplicand. The pairs in *alist* are ordered according to the multiplicands, with the heaviest first, and no two distinct pairs have identical multiplicands. The multiplicand in the first pair of the *alist* (the heaviest) is called the *key multiplicand* of the polynomial and the *sign* of the polynomial is the sign of the key multiplicand's coefficient. The third field of a polynomial, *hyps*, contains a set of terms. The fourth and fifth fields, *lits* and *supports*, contain sets of LISP objects. (Note: the last three fields all contain LISP lists treated as sets. However, we use EQUAL to compare elements in the hyps field but EQ to compare elements in the other two fields.)

The *formula represented* by a polynomial with constant *c*, *alist* $(\langle t_1, k_1 \rangle, \ldots, \langle t_n, k_n \rangle)$, and *hyps* $h_1, \ldots, h_m$ is

$$h_1 \wedge \cdots \wedge h_m \rightarrow c + k_1 * t_1 + \cdots + k_n * t_n \leq 0.$$

The *lits* field of a polynomial contains the literals linearized to produce the polynomial or its ancestors. The *hyps* of a polynomial contain the linearization hypotheses. The supports field contains a variety of things: the names of the axioms, definitions, and lemmas used in the derivation of the polynomial, the literals used to relieve linearization hypotheses or the hypotheses of rewrite and linear rules contributing to the derivation of the polynomial, and special marks explained in Section 8.7.

We say a polynomial is *impossible* iff its constant is greater than 0 and no coefficient is negative. We say a polynomial is *vacuous* iff its constant is less than or equal to 0 and no coefficient is positive. Observe that if a polynomial is impossible the conclusion of the formula it represents is contradictory. Similarly, if a polynomial is vacuous then the conclusion of its formula is trivially true.

## 8.3. Converting terms to polynomials

The process of converting a literal into one or more polynomials is called 'linearization'. Linearization implicitly takes place in a context (as does rewriting). The linearization of *lit* is either NIL or a set of sets of polynomials. If the result is NIL, we draw no arithmetic conclusions from assuming *lit*. Otherwise, the answer represents a formula, *form*, obtained by disjoining (across the set) the result of conjoining (across each element) the formulas represented by each polynomial. It is a theorem that *form* is implied by the assumptions in the context.

Below we show some examples of literals linearized and the formulas represented by the answers. In each of the cases below, the *lits* field of the polynomials returned is $\{lit\}$ and the support fields is $\{\}$

| *lit* | formula represented by result of linearization |
|---|---|
| $I<J\dot{-}I$ | $I\leqslant J\rightarrow 1+-1*J+2*I\leq 0$ |
| $I=J\dot{-}I$ | $I\leqslant J\rightarrow 0+-1*J+2*I\leq 0$ |
| $I\neq J\dot{-}I$ | $0+1*J+-2*I\leq 0$ |
| | $I\leqslant J\wedge I\in\mathcal{N}\rightarrow 1+-1*J+2*I\leq 0$ |
| | $I\in\mathcal{N}\overset{\vee}{\rightarrow}1+1*J+-2*I\leq 0.$ |

In order to describe the linearization process we need three auxiliary concepts. The first is the notion of the *zero polynomial depending on lit*, which is the polynomial with constant 0, empty *alist, hyps,* and *supports* fields, and *lits* field $\{lit\}$.

The other two concepts we mention informally here and define precisely after discussing linearization. One concept is that of 'inserting a hypothesis *hyp* into a polynomial *p*', which, roughly speaking, is the construction of a polynomial identical to *p* except that *hyp* is included in the hyps field. The third notion is that of 'adding a term *t* positively (or negatively) to a polynomial $p'$. Roughly speaking, this means constructing a new polynomial by adding $t$ to (or subtracting $t$ from) $p$. However, when we add a term to a polynomial we may also insert hypotheses.

Careful treatment of the hypotheses is essential to the utility of our linear arithmetic procedure. Omitting a hypothesis that is not known to be true can cause unsoundness. But failure to recognize that a hypothesis is already known to be true or false can cause unnecessary case splitting or infinite looping (as the system may case split repeatedly on the same condition). Exactly how the linearization procedure handles the hypotheses depends upon how the procedure is being used.

It is useful to distinguish two different occasions on which we linearize terms. The first, and simpler of the two, is during the process of rewriting a literal after we have set up our polynomial data base. Linearization is used both to relieve hypotheses of rewrite rules and to augment the data base. But the data base produced by pushing the polynomials is not saved—we are only looking for a contradiction. Hence we need not track our dependency on literals and can use the type alist, TA, supplied by the context of the rewriter, to check the truth or falsity of some linearization hypotheses. If a hypothesis is true under TA we need not include it. If it is false we should avoid producing a polynomial requiring its truth.

The second occasion we use linearization is when we are setting up the initial data base. In this case we must track our dependencies on literals very carefully to avoid tail biting. During the initialization of the data base we therefore use a context in which TA is empty—preventing the unreported use of type set information—and use LITS-THAT-MAY-BE-ASSUMED-FALSE and HEURISTIC-TA to determine the truth or falsity of some linearization hypotheses. For example, if the complement of a required linearization hypothesis is in LITS-THAT-MAY-BE-ASSUMED-FALSE (which, recall, is the clause being proved), then we need not include it in the hyps field of the polynomials but must include it in the supports field.

Looking for a hypothesis or its complement in LITS-THAT-MAY-BE-ASSUMED-FALSE is not as powerful as computing its type set. For example, the type set mechanism could deduce the truth of (NUMBERP $t$) from the assumption $t = A \oplus B$. Nevertheless, we have adopted this approach because we must know which literals in the clause are being used when a required linearization hypothesis is omitted. However, recall that if we believe a hypothesis is false we simply avoid producing a polynomial requiring its truth. The soundness of the theorem prover is unaffected by the validity of our belief that a hypothesis is false: at worst we deny the system access to information it could have used. In this case it is irrelevant to track dependencies, since no polynomial is produced. Thus, we can afford to use a type alist as a heuristic device to avoid the production of certain polynomials. This is the role of HEURISTIC-TA in the context and it encodes the negations of all the literals of the clause.

Except where noted, all type set computations are done with respect to TA.

We say a term $t$ is *possibly numeric* if the type set of $t$ under HEURISTIC-TA (or, if HEURISTIC-TA is NIL, under TA) is {NUMBERP}.

The *positive (or negative) linearization of a literal lit* is either NIL or a set of sets of polynomials as described below. In the description below we handle the positive case only. The negative case is identical to the positive case for the complement of *lit* with one exception: the lits field of all the polynomials constructed contain *lit* rather than its complement.

If any polynomial in any element of the answer contains $F$ in its hyps field we delete that polynomial from the element.

If *lit* is of the form (LESSP *lhs rhs*) the answer is $\{\{poly\}\}$ where *poly* is the result of adding the term (ADD1 *lhs*) positively to the result of adding the term r.h.s. negatively to the zero polynomial for *lit.*

If *lit* is of the form (EQUAL *lhs rhs*) and either *lhs* or *rhs* is possibly numeric, the answer is $\{\{poly_1 poly_2\}\}$ where $poly_1$ is the result of adding *lhs* positively to the result of adding r.h.s. negatively to the zero polynomial for *lit,* and $poly_2$ is obtained by the symmetric procedure (swapping the roles of *lhs* and *rhs*).

If *lit* is (NOT (LESSP *lhs rhs*)) the answer is $\{\{poly\}\}$ where *poly* is the result of adding *rhs* positively to the result of adding *lhs* negatively to the zero polynomial for *lit.*

If *lit* is (NOT (EQUAL *lhs rhs*)) and either *lhs* or *rhs* is possibly numeric, then let $poly_1$ be the result of inserting (NUMBERP *lhs*) and (NUMBERP *rhs*) hypotheses into the result of adding (ADD1 *lhs*) positively to the result of adding *rhs* negatively to the zero polynomial for *lit,* and let $poly_2$ be obtained by the symmetric procedure (swapping the roles of *lhs* and *rhs*). If $poly_1$ is impossible, the answer is $\{\{poly_2'\}\}$ where $poly_2'$ is obtained from $poly_2$ by adding to its *hyps* field those of $poly_1$; if $poly_2$ is impossible, the answer is $\{\{poly_1'\}\}$ where $poly_1'$ is obtained from $poly_1$ by adding to its *hyps* field those of $poly_2$; otherwise the answer is $\{\{poly_1\}\{poly_2\}\}$.

If none of the above four cases obtains, the answer is NIL.

This concludes the definition of linearization.

The result of adding a term to a polynomial involves manipulating the alist of the polynomial (and possibly the hyps field). The following subsidiary concept is used:

The result of *inserting a term t with a coefficient of n into the alist field of a polynomial poly* is the polynomial that results from *poly* by modifying its alist as follows. If the type set of $t$ does not include NUMBERP, do not modify the alist (since our arithmetic functions coerce non-NUMBERP arguments to 0); if there is a pair with multiplicant $t$ in the alist, increment the coefficient of that pair by $n$; otherwise, add the pair $\langle t, n\rangle$ to the alist (maintaining the previously noted ordering of entries).

The result of *adding a term t with parity p to a polynomial poly* is the polynomial obtained as follows:

If *t* is a constant, then if *t* is a natural number, increment (decrement) the constant of *poly* by *t* (according to whether *p* is positive or negative) and return the result; otherwise return *poly* (since non-NUMBERPS are coerced to 0).

If *t* is (ADD1 *x*), increment (decrement) the constant in *poly* by 1 (according to whether *p* is positive or negative) and add *x* with parity *p* to the result.

If *t* is (SUB1 *x*), then if *p* is positive: decrement the constant in *poly* by 1 and add *x* with parity *p* to the result; otherwise *p* is negative: insert the hypothesis (NOT (LESSP *x* 1)) into *poly*, increment the constant in the resulting polynomial by 1, and add *x* with parity *p* to the result.

If *t* is (PLUS *x y*), add *y* with parity *p* to the result of adding *x* with parity *p* to poly.

If *t* is (DIFFERENCE *x y*), then if *p* is positive: add *x* positively to the result of adding *y* negatively to *poly*; otherwise *p* is negative: insert the hypothesis (NOT (LESSP *x y*)) into *poly* and then to the result add *x* negatively to the result of adding *y* positively.

If *t* is (TIMES *n x*), where *n* is a natural number, insert *x* with a coefficient of *n* (or $-n$) (according to whether *p* is positive or negative) into the alist field of *poly* and return the result.

Otherwise, insert *t* with a coefficient of 1 (or $-1$) (according to whether *p* is positive or negative) into the alist field of *poly* and return the result.

This completes the definition of how to add a term to a polynomial.

The result of *inserting the hypothesis hyp into a polynomial poly* is obtained as follows.

If *hyp* is (NOT (LESSP *x* 1)) and the type set of *x* is {NUMBERP}, insert the hypothesis (NOT (EQUAL *x* 0)) into *poly* instead and return the result.

If *hyp* is (NOT (LESSP *x* 1)) and the complement of (NUMBERP *x*) occurs as some literal lit in LITS-THAT-MAY-BE-ASSUMED-FALSE, add lit to the supports field of *poly* and return the result.

If *hyp* is (NOT (EQUAL (DIFFERENCE *u v*) 0)), insert the hypothesis (LESSP *v u*) into *poly* and return the result.

If *hyp* is (NOT (EQUAL (ADD1 *x*) 0)), return *poly*.

If *hyp* is (NOT (EQUAL *n* 0)) where *n* is any constant other than 0, return *poly*.

If *hyp* is (NOT (EQUAL 0 0)), insert the hypothesis *F* into *poly* and return the result.

If the type set of *hyp* is {TRUE}, return *poly*.

If the type set of *hyp* does not include TRUE, insert the hypothesis *F* into *poly* and return the result.

If the type set of *t* (computed with HEURISTIC-TA) does not include TRUE, insert the hypothesis *F* into *poly* and return the result.

If the complement of *hyp* occurs as some member, *lit*, of LITS-THAT-MAY-

BE-ASSUMED-FALSE, add *lit* to supports field of *poly* and return the result.

Otherwise, add *hyp* to the hypothesis field of *poly* and return the result.

## 8.4. Combining polynomials

Suppose $p_1$ and $p_2$ are polynomials with the same key multiplicand, $t$, and opposite signs. Let the coefficients of $t$ in $p_1$ and $p_2$ be $k_1$ and $k_2$, respectively. By *cross-multiplying and adding* $p_1$ and $p_2$ we can form a new polynomial whose key multiplicand, if any, is smaller than $t$. The polynomial obtained has as its constant $k_2 * c_1 + k_1 * c_2$, where $c_1$ and $c_2$ are the constants of $p_1$ and $p_2$, respectively. The alist of the new polynomial is obtained from the alists of $p_1$ and $p_2$ by multiplying each coefficient in the first by $k_2$ and each coefficient in the second by $k_1$, then merging the two alists (adding together the coefficients of identical multiplicands and deleting any pair with a 0 coefficient). The hyps, lits, and supports fields of the new polynomial are the unions of the corresponding fields of $p_1$ and $p_2$ (comparing with EQUAL or EQ as appropriate).

Observe that if the formulas represented by $p_1$ and $p_2$ are both true in the context then so is the formula represented by the result of cross-multiplying and adding.

For each term $t$ we define the *non-negative assumption* for $t$ to be the polynomial obtained by linearizing the theorem $0 \leqslant t$, i.e. the polynomial representing $0 + -1 * t \leq 0$, with empty hyps, lits, and supports fields. Any polynomial with a positive first coefficient can be cross-multiplied and added to the appropriate non-negative assumption to obtain a true polynomial differing from the initial polynomial only in that the first pair in the alist is missing.

## 8.5. Pushing polynomials into the data base

Conceptually, our linear arithmetic data base is just a set of polynomials. To make it easier to find all the polynomials with a given key multiplicand and sign, we actually partition the data base into 'pots' according to their key multiplicands and further partition each pot according to the sign of the polynomials. We then store the pots in order according to the weight of the key multiplicands. However, for the purposes of this paper we treat the data base simply as a set of polynomials.

The fundamental operation on the data base is to add new polynomials to it and deduce the consequences by cross-multiplying and adding. However, recall that during the simplification of a given literal we wish not to use polynomials that descend from LITS-TO-BE-IGNORED-BY-LINEAR. We say a polynomial poly is *available* if no element of LITS-TO-BE-IGNORED-BY-LINEAR is EQ to any element of the lits or supports fields of poly.

The result of *pushing a set of polynomials s into a data base db* is the closure of the union of *db* and *s* under the following two operations:

1. For any available member polynomial *x* with positive sign, include the result of cross-multiplying and adding *x* to the non-negative assumption for the key multiplicand of *x*, provided that result is non-vacuous.

2. For any two available member polynomials *x* and *y* with the same key multiplicand and opposite signs, include the result of cross-multiplying and adding *x* and *y*, provided that result is non-vacuous.

The above description fails to describe our code in three respects. First, because the initial data base is closed under the operations above, it suffices to consider only the new polynomials and their consequences. Second, the order in which we combine polynomials is not specified. Third, since we are seeking to derive an impossible polynomial the code that closes the data base halts when a cross multiply and add produces an impossible polynomial. The hyps, lits, and supports fields of the impossible polynomial found influence the subsequent proof attempt. Thus, if more than one impossible polynomial can be derived from the assumptions, the order in which polynomials are combined is relevant.

### 8.6. Augmenting the data base

In this subsection we explain how we use previously proved theorems to augment the data base of polynomials.

A *linear rule* is a four-tuple ⟨*name, hyps, concl, max-term*⟩, where *name* is the user-supplied name of a formula, *hyps* is a list of terms, *concl* is a term of the form (LESSP *x y*) or (NOT (LESSP *x y*)), the positive linearization of *concl* (under empty TA, LITS-THAT-MAY-BE-ASSUMED-FALSE, and HEURISTIC-TA) is a singleton set containing a singleton set containing a polynomial *poly*, and *max-term* is one of the multiplicands in the *alist* of *poly* and has the following properties: (a) *max-term* is not a variable symbol; (b) the set of variables occurring in *concl* is a subset of the union of those occurring in *max-term* and those occurring in *hyps*; and (c) no other multiplicand in the *alist* of *poly* has larger size and contains a superset of the variables occurring in *max-term*.

Roughly speaking, linear rules are interpreted as follows. Whenever a new key multiplicand is introduced into the polynomial data base we search for applicable linear rules, finding each rule whose max-term can be instantiated to yield the key multiplicand in question. When we find such a rule we attempt to establish, by rewriting, the corresponding instance of each of the hypotheses in the hyps of the rule. Provided we succeed, we rewrite the appropriate instance of the concl of the rule and linearize it to obtain a polynomial. We then modify the hyps and supports fields of the polynomial to take into account the hypotheses assumed and lemmas and literals used during the rewriting. Then, provided certain heuristic conditions are met, we push the resulting polynomial into the data base.

The restrictions on max-term above are motivated by two considerations. First, we want to ensure that once the variables in a maximal term are instantiated (by the pattern match with a key multiplicand) and the hypotheses are relieved (possibly instantiating variables occurring in hyps but not in max-term), then every variable in concl is instantiated. Second, since the polynomial produced from the instantiated conclusion can only be used to cancel its heaviest multiplicand, we try to select as our max-terms only those terms which might, under suitable instantiation, become the largest.

Linear rules are added to the system's library of rules whenever certain user-supplied formulas are proved. Suppose the user submits to the theorem prover a conjecture named *name* of the form (IMPLIES *hyp concl*). Suppose further the conjecture was tagged as a rewrite rule. Let *hyps* be the result of flattening the AND structure of *hyp*, i.e. the conjunction over *hyps* is *hyp*. If the conjecture is proved, we store in our library each four-tuple ⟨*name, hyps, concl, t*⟩ that is a linear rule. Actually, the recognition of candidate theorems is more sophisticated. For example, a simple (LESSP *x y*) or (NOT (LESSP *x y*)) theorem is recognized as a candidate and *hyp* defaults to *T*. If *concl* is a conjunction, we strip out the individual conjuncts and look for inequalities. These details are unimportant in this paper.

Before linearizing the instantiated conclusion of a linear rule we rewrite it to put the terms into normal form under the current set of rewrite rules. However, rather than rewrite the entire conclusion, we rewrite merely the two sides of the inequality to avoid applying linear arithmetic to the conclusion before we have normalized the terms.

The *rewritten form of term under substitution s*, where *term* is a term of the form (LESSP *lhs rhs*) or (NOT (LESSP *lhs rhs*)) is obtained as follows. Let *lhs'* and *rhs'* be obtained by rewriting *lhs* and *rhs*, respectively, under the substitution *s*. If *concl* is (LESSP *lhs rhs*), the rewritten form is (LESSP *lhs' rhs'*), otherwise, it is (NOT (LESSP *lhs' rhs'*)).

*Pushing a linear rule ⟨name, hyps, concl, max-term⟩ for multiplicand t into a data base db* produces a data base as follows. If *db* contains an impossible polynomial, return *db*. If there is no substitution *s* on the variables of *max-term* such that *s* applied to *max-term* is *t*, return *db*. Otherwise, push new frames onto both the lemma stack and the hyps stack, and, using *db* as DB, attempt to relieve the hypotheses *hyps*. This either fails or succeeds and delivers an extension *s'* of *s* and modifies the top frames of the two stacks. If the attempt fails, pop and discard the two frames added and return *db*. Otherwise, pop the lemma stack and let lemmas be the resulting set of items. Pop the hyps stack and let *hyps* be the resulting set of terms. Let {{*poly*}} be the positive linearization of the rewritten form of concl under $s'$. If for any reason the linearization does not produce such a structure or if there is a multiplicand in the alist of *poly* that is distinct from, as large as, and 'worse than' every key multiplicand in *db*, then return *db*. (We use the same sense of 'worse than' defined on p. 110 of Boyer and Moore, 1979.) Otherwise, let *poly'* be obtained from *poly* by setting the hyps field to the union of the hyps of *poly* and *hyps*, and setting the supports field to the union of {*name*} and lemmas. Return the result of pushing *poly* into *db*.

The result of *augmenting a data base db with linear rules for a set of multiplicands s* is a polynomial data base constructed as follows. If *db* contains an impossible polynomial, return *db*. If *s* is empty, return *db*. Otherwise, let *db'* be the result of iteratively expanding *db* by pushing into it each linear rule about any multiplicand in *s*. Return the result of augmenting *db'* with linear rules for every non-variable key multiplicand in *db'* that is not a key multiplicand in *db*.

The resulting of *pushing a set of polynomials s into a data base db and augmenting with linear rules* is the data base constructed as follows. Let *db'* be the result of pushing *s* into *db*. Return the result of augmenting *db'* with linear rules for all non-variable key multiplicands in *db'* that are not key multiplicands in *db*.

## 8.7. The interface between linear arithmetic and rewriting

In this subsection we describe the operation of pushing terms (as opposed to polynomials) into a data base and augmenting with lemmas. This operation is the entry to the linear arithmetic procedure from the rest of the simplifier. It is used both to construct the initial data base and to rewrite inequalities by showing they contradict our previous assumptions.

Given a data base encoding our current linear assumptions and a list of terms to assume true (or false) we desire to construct a new data base containing the conjunction of the old and new assumptions. If each term linearized into a conjunction of polynomials the task would be simple: linearize each term, push each polynomial produced and then augment the data base with linear rules. However, some terms, e.g. $I \neq J$, linearize to a disjunction of polynomials: either $I<J$ or $J<I$. A single data base cannot, in general, represent the assumption $I \neq J$. However, if $I<J$ contradicts other assumptions, we can push $J<I$, and vice versa. Our initial implementation simply ignored disjoined polynomials, but we found several cases where that prevented proofs. We dismissed as too expensive (without even implementing it) the much stronger approach of producing a data base for each combination of alternatives and carrying out the desired simplifications in each of them.

The result of *pushing the list of terms s positively* (*or negatively*) *into the data base db and augmenting with linear rules* is the data base obtained as follows: linearize each term in *s* (positively or negatively, as indicated). Each answer can be classified into one of three categories: it is a singleton list containing a list of polynomials, in which case we say the polynomials are *conjuncts*; it is a doubleton list containing two lists of polynomials, in which case the doubleton is said to be a *pair of alternatives*; or it is neither of the above, in which case the linearized term was not recognized as an arithmetic equality or inequality. Let *db'* be the result of pushing all of the conjunct polynomials into *db* and augmenting with linear rules. Then, iteratively expand *db'* by considering each pair of alternatives $\{poly\text{-}lst_1\ poly\text{-}lst_2\}$ and doing the following: if the result of pushing $poly\text{-}lst_1$ into *db'* and augmenting with linear rules contains an impossible polynomial, modify the hyps and supports fields of the polynomials in $poly\text{-}lst_2$ by unioning into them the hyps and supports fields (respectively) of the impossible polynomial found, and then replace *db'* by the result of pushing the modified $poly\text{-}lst_2$ into *db'* and augmenting with linear rules; otherwise (if pushing $poly\text{-}lst_1$ produced no contradiction), perform the symmetric test with $poly\text{-}lst_2$ and modify and push $poly\text{-}lst_1$ if a contradiction is found; otherwise, do not expand *db'* on this iteration. When all alternatives have been considered, return the final *db'*.

As described above the consideration of the alternatives is needlessly expensive: if pushing $poly\text{-}lst_1$ into *db'* does not lead to a contradiction but pushing $poly\text{-}lst_2$ does, we push $poly\text{-}lst_1$ into *db'* again after modifying its hyps and supports. Of course, the data base produced by pushing the modified $poly\text{-}lst_1$ is exactly the same as that produced by pushing $poly\text{-}lst_1$ except that the consequences derived from the modified $poly\text{-}lst_1$ have additional hyps and supports. But pushing and augmenting can be quite expensive since it causes conditional rewriting and backwards chaining. Our implementation avoids the near-duplicated pushes by putting a unique mark in the supports field of $poly\text{-}lst_1$ before it is pushed the first time. Because of the way the supports field is propagated by cross-multiplication and adding, every consequence deduced from members of $poly\text{-}lst_1$ is marked in the resulting data base, $db'_1$. If $db'_1$ does not contain a contradiction but pushing $poly\text{-}lst_2$ into *db'* does, we visit every marked polynomial in $db'_1$ and update the hyps and supports fields with those from the contradiction found with $poly\text{-}lst_2$.

### 8.8. Rewriting terms and relieving hypotheses

We now complete our description of how the rewriter has been modified to use linear arithmetic.

At the point in rewriting where we used to 'rewrite with lemmas' (p. 122 of Boyer and Moore, 1979) we now try linear arithmetic first, provided the atom of the term being rewritten is a LESSP or EQUAL expression and the objective of rewriting is either to show the term T or to show it F. If the objective is to show the term T, we push the term negatively into DB and augment with linear rules. If the resulting data base contains an impossible polynomial, *poly*, we add to the top frame of the *hyps* stack the terms in the *hyps* field of *poly*, add to the top frame of the lemma stack the items in the *supports* field of poly, and return T as the value of the rewritten term. If, on the other hand, the objective is to show the term *F*, we do the symmetric operation.

In an earlier implementation we tried using linear arithmetic to simplify LESSP or EQUAL terms even when the objective was not T or F. In particular, we first tried pushing the term positively and if that produced no contradiction, we tried pushing it negatively. To our surprise, this increased the total number of conses used during the proofs of the theorems in Appendix A of Boyer and Moore (1979) from roughly 6 million to roughly 10 million without significantly shortening the proofs produced. We therefore abandoned the idea of using linear arithmetic except when we had a clear objective to establish.

The remaining changes to the rewriter are motivated by the need to track accurately which literals are being used when we augment the initial data base with linear rules. To prevent surreptitious use of type information, we set TA to NIL during the construction of the data base. This cripples the rewriter described in Boyer and Moore (1979) since it has no assumptions with which to work while trying to relieve the hypotheses of linear rules. We use LITS-THAT-MAY-BE-ASSUMED-FALSE to encode assumptions in a way that permits us to track dependencies.

As noted on p. 124 of Boyer and Moore (1979), just before the rewriter returns its answer, *ans*, it asks whether *ans* has typeset {**TRUEP**} or {**FALSEP**} under TA and, if so, returns T or F, as appropriate, instead. Now we ask, in addition, whether *ans* is EQUAL to some member, *lit*, of LITS-THAT-MAY-BE-ASSUMED-FALSE. If so, we return F instead, but we add LIT to the top frame of the lemma stack. That literal will ultimately be deposited in the supports field of any polynomial depending on this rewrite. Similarly, if the complement of *ans* occurs in LITS-THAT-MAY-BE-ASSUMED-FALSE we return T instead and store the corresponding *lit* in the lemma stack, provided that *ans* is Boolean valued or that the sense of equality to be preserved by this rewrite is propositional equivalence.

As noted on p. 122 of Boyer and Moore (1979), when we are trying to relieve a hypothesis *hyp* under some substitution *s* and *s* does not instantiate every variable of *hyp* we use TA to try to extend *s* to make the instantiation of *hyp* true. We now use LITS-THAT-MAY-BE-ASSUMED-FALSE in an analogous way, recording on the lemma stack the literals used. In addition, if any hypothesis to be established is on LITS-THAT-MAY-BE-ASSUMED-FALSE we abandon the attempt to relieve the hypotheses.

**8.9. Deriving equalities from the data base**

In this subsection we define the concepts necessary to describe how we generate from the polynomial data base equality literals to add to the clause being proved.

We say a polynomial $p$ *isolates* $t_i$ *positively* (*or negatively*) iff $t_i$ is a multiplicand in the *alist* of $p$, the coefficient, $k_i$, of $t_i$ is positive (negative), the constant of $p$ and all coefficients other than that of $t_i$ are negative (positive) and multiples of $k_i$, and the lits field of $p$ is not a singleton set containing a negated equality.

Note that if a polynomial with constant $c$ and *alist* $(\langle t_1, k_1\rangle, \ldots, \langle t_n, k_n\rangle)$ isolates $t_i$ positively then the concluding inequality in the formula represented by the polynomial can be put into the form:

$$t_i \leq c' + k'_1 * t_1 + \cdots + k'_{i-1} * t_{i-1} + k'_{i+1} * t_{i+1} + \cdots + k'_n * t_n,$$

where $c'$ and the $k'_i$'s are all natural numbers. We call $t_i$ the *isolated term* of the polynomial and

```
(PLUS c'
        (TIMES $k'_1 t_1$) ···
        (TIMES $k'_{i-1}\ t_{i-1}$)
        (TIMES $k'_{i+1}\ t_{i+1}$) ...
        (TIMES $k'_n\ t_n$))
```

the *conglomerated term corresponding to t.*

The *result of multiplying* (*or dividing*) *a polynomial poly by an integer n* is a five-tuple <$c$, *alist, hyps, lits, support* >, where $c$ is the constant of *poly* multiplied (or divided) by $n$, *alist* is obtained from the alist of *poly* by multiplying (or dividing) each coefficient by $n$, and the remaining fields are those of the same name in *poly*. Multiplying a polynomial by $n$ produces a polynomial. Dividing a polynomial by $n$ produces a polynomial only if $n$ divides the constant and each coefficient.

We say a polynomial $poly_2$ is a *complementary multiple* of a polynomial $poly_1$ iff there is a negative integer $n$ such that the result of multiplying $poly_1$ by $n$ is $poly_2$.

We say two polynomials, $poly_1$ and $poly_2$ are *mates on a term t* iff $poly_1$ isolates $t$ (positively or negatively) and $poly_2$ is a complementary multiple of the result of dividing $poly_1$ by the coefficient of $t$ in $poly_1$.

If a data base contains two mates, $poly_1$ and $poly_2$, on a term $t$ then, under the conjunction over the union of the hypotheses in the two polynomials, we can derive an equation between $t$ and its corresponding conglomerated term. In the next subsection we describe how we process mated polynomials.

## 8.10. Simplifying clauses

Roughly speaking, to simplify a clause we first set up a polynomial data base derived by assuming all the literals of the clause false. If the data base contains an impossible polynomial we are done. Otherwise, we look for mated polynomials and process them. If we find no mates, we sweep the clause from left to right rewriting each literal in turn, using the polynomial data base previously set up but ignoring certain polynomials in it. At each stage we must deal with the linearization hypotheses arising from polynomials we have used.

The *polynomial data base for the clause cl* is constructed as follows. First, we bind LITS-THAT-MAY-BE-ASSUMED-FALSE to *cl,* HEURISTIC-TA to the type alist encoding the falsity of every term in *cl,* TA to NIL, and LITS-TO-BE-IGNORED-BY-LINEAR to NIL. Then we push *cl* negatively into the empty data base and augment it with linear rules and return the result.

If a clause is a consequence of simple linear arithmetic, the polynomial data base will contain an impossible polynomial. However, because TA is NIL during the augmentation of the polynomial data base we sometimes fail to find contradictions (involving linear rules) that would be found if TA contained the negations of all the literals. Therefore, when we simplify a clause we take time out to augment the data base under the stronger TA, hoping to generate an impossible polynomial. If no contradiction is found we discard the resulting data base since it contains 'hidden' dependencies.

The control structure of clause simplification exploits the fact that clauses are represented by sequences, not sets. In addition, we must agree upon a way to mark the 'current literal' in a clause. We will continue to use set brackets to denote clauses but will consider the objects described to be sequences and will attach importance to the order of the literals. When we use the 'union' operator, $\cup$, in connection with clauses, we mean concatenation. The 'current literal' of a clause will be enclosed in square brackets. Thus, $\{\neg p\ [q]\ r\}$ is a clause whose first literal is $\neg p$ and whose current literal is $q$.

The result of *splitting the clause* $\{\ldots p\ [q]\ r \ldots\}$ *on* $h_1, \ldots, h_n$ is the set consisting exactly of each clause of the form $\{\ldots p\ h_i\ q\ [r] \ldots\}$, where $1 \leq i \leq n$. If there is no literal $r$ to the right of the selected literal $q$ in the input, the clauses in the output set have no selected literal.

The result of *adding the hypotheses* $h_1, \ldots, h_n$ *to the clause* $\{\ldots p\ [q]\ r \ldots\}$ is the clause $\{\ldots p\ \neg h_1 \ldots \neg h_n\ [q]\ r \ldots\}$.

The result of *splicing the clause segments* $seg_1, \ldots, seg_n$ *in place of the selected literal in* $\{\ldots p\ [q]\ r \ldots\}$ is the set consisting exactly of the clauses $\{\ldots p\} \cup seg_i \cup \{[r] \ldots\}$, where $1 \leq i \leq n$. If there is no literal $r$ to the right of the selected literal $q$ in the input, the clauses in the output set have no selected literal.

We now define the heuristic for controlling the introduction of derived equalities and the way we handle the hypotheses generated by the derivation.

The *heuristics for equality introduction* for two terms $t$ and $t'$ is the condition that the type set of both $t$ and $t'$ contains NUMBERP and that no clause in the ancestry of the clause being simplified is a 'result of adding the equation of $t$ and $t'$' as defined below. (Every clause processed by the theorem prover comes with a complete history of its derivation, including its parent and the operations that produced it.)

The result of *introducing into a clause cl the equality between t and t' derived from two polynomials poly*$_1$ *and poly*$_2$ is the union of $S_1$ and $S_2$ defined below. Let *hyps* be the result of unioning together the hyps fields of the two polynomials and then adding the term (NUMBERP $t$) (unless the type set of $t$ is {NUMBERP}) and the term (NUMBERP $t'$) (unless the type set of $t'$ is {NUMBERP}). $S_1$ is the singleton set containing the result of adding the hypothesis $t = t'$ to the result of adding the hypotheses *hyps* to *cl*. $S_2$ is the result of splitting *cl* on *hyps*. We say every clause in $S_1$ and $S_2$ is a *result of adding the equation of t and t'*.

To *sweep a clause cl*, $\{\ldots\ p\ [q]\ r\ \ldots\}$, construct a set of clauses as follows. If there is no selected literal, return $\{cl\}$. Otherwise, let TA be the type alist obtained by assuming false every literal in *cl* except the selected literal, $q$. Let LITS-THAT-MAY-BE-ASSUMED-FALSE and HEURISTIC-TA be NIL. Let LITS-TO-BE-IGNORED-BY-LINEAR be the list containing $q$ and every literal to its left in CL, that 'rewrote to F' as defined below. Push empty frames onto both the hyps stack and the lemma stack. Let $q'$ be the result of rewriting $q$. If $q'$ is F, we say $q$ *rewrote to* F. Pop and discard the top frame of the lemma stack. (Actually, the names in that frame are accumulated and eventually printed as part of a description of the proof. In addition, they are used to build a dependency graph when the system's library is updated at the end of successful proofs.) Pop the top frame of the hyps stack and let *hyps* be the set of terms in that frame.

Let *segs* be the set of clause segments obtained by normalizing the IFS in $q'$ and splitting out each branch, as shown on page 124 of Boyer and Moore (1979). For example, if $q'$ is ($G$ (IF $a$ $b$ $c$)) then we obtain two clause segments $\{\neg a\ (Gb)\}$ and $\{a\ (Gc)\}$. The final answer is obtained by recursively sweeping each clause in the union of $S_1$ and $S_2$ (defined below) and unioning together the results. $S_1$ is the result of splicing *segs* in place of the selected literal in the clause obtained by adding *hyps* to *cl*. $S_2$ is the result of splitting *cl* on *hyps*.

To *simplify a clause, cl,* construct a set of clauses as follows. Select the first literal of *cl* as the current literal. Let DB be the polynomial data base for *cl*. If there is an impossible polynomial, *poly,* in DB, return the result of splitting *cl* on the hypotheses of *poly*. Otherwise, let TA be the type *alist* obtained by assuming all literals of *cl* false. If there is an impossible polynomial, *poly*, in the result of augmenting DB with linear rules for every key multiplicand in DB, return the result of splitting *cl* on the hypotheses of *poly*. If there are two polynomials in DB that are mates on some term *t* with conglomerated term *t'* and the heuristics for equality introduction are satisfied, return the result of introducing into *cl* the equality between *t* and *t'* derived from the two polynomials. Otherwise, sweep *cl* and return the result.

## 9. EFFICIENCY

The incorporation of the linear procedure sketched above has dramatically improved the performance of our theorem prover on arithmetic problems. For example, compared to the theorem prover described in Boyer and Moore (1979) the system spends 40% less time processing the theorems and definitions in Appendix A of Boyer and Moore (1979). Furthermore, we have been able to eliminate the need for the user to state explicitly linear facts. Thus, our original objective was achieved. Among the theorems proved by the latest version of the theorem prover are the invertibility of the Rivest, Shamir, and Adleman public key encryption algorithm (Boyer and Moore, 1984), Wilson's theorem (Rusinoff, 1983), Gauss's law of quadratic reciprocity (David M. Rusinoff led the theorem prover to Gauss's law) and the Church-Rosser theorem (Shankar, 1985). All of these proofs involved a substantial amount of linear arithmetic reasoning.

We now turn to our observation that theoretical efficiency is not a good measure of the utility of a linear procedure in a larger system. Let us reconsider our decision to use the simple 'cross-multiply and add' algorithm instead of more efficient ones. Might our handling of arithmetic be sped up by the use of another propagation algorithm? The answer is no; an insignificant portion of the time is devoted to the problem of propagating polynomials through the data base. Consider what else must be done. Terms must be linearized, the key multiplicands in the data base determined, interesting lemmas must be selected and instantiated (and their hypotheses must be relieved by nonarithmetic reasoning) with due caution for avoiding traps like 'pumps', the lemmas, literals, and hypotheses supporting the derivation of each inequality must be recorded and maintained, one must avoid biting one's own tail, one must be able to 'pop' or 'undo' the effect of pushing an inequality and all of the linear rules it introduced, and when a linear contradiction is found one must handle the additional cases raised by the particular contradiction found.

Consider the MS proof sketched above. Of the total time spent in arithmetic reasoning in the second phase of the proof (5.8 seconds) only 1.8% (0.119 seconds) is spent propagating polynomials. The rest is spent taking care of the issues listed above. Thus, the availability of an instantaneous oracle for linear arithmetic problems would speed up the MS proof by an insignificant amount.

Perhaps more realistic data is that obtained during the proof of the verification conditions for the FORTRAN version of our fast string searching algorithm. We regard this set of 53 lemmas and verification conditions to be quite representative of the verification of practical programs. The verification conditions establish that the preprocessor correctly sets up a global COMMON array and that the search algorithm correctly computes the location of the first occurrence of the pattern in the text, if there is an occurrence, or else correctly announces that no occurrence exists. Furthermore, the v.c.s establish that there are no array bounds violations, arithmetic overflows, or other run time errors, and that both subroutines terminate. To prove these v.c.s the system must first establish several important lemmas about strings and string searching. These lemmas are proved inductively from the definitions of such concepts as 'a string over a finite alphabet', 'leftmost occurrence' and our DELTA1 function. The definitions themselves are proved satisfiable by the system before they are admitted. The admission of the definitions, proofs of the lemmas, and proofs of the v.c.s all require both arithmetic and non-arithmetic reasoning, as is common in the verification of programs that compute non-arithmetic functions on arrays and tables (e.g. searching, sorting, hashing).

The total time taken is 1417 c.p.u. seconds (23.6 c.p.u. minutes). We push terms into the polynomial data base 2637 times. Relatively few linear rules are available for instantiation. Only once in every six calls does the augmentation procedure find a lemma that is judged to be relevant to the data base. (Thus, one can infer that not an inordinate amount of time is spent pursuing instantiations.) The total time consumed while pushing terms into the data base is 357 c.p.u. seconds, 25% of the total proof time. But only 38.5 seconds is spent pushing polynomials. That is, in this fairly representative verification problem, an instantaneous oracle for linear inequalities would reduce the time in arithmetic reasoning by 10.7% and would reduce the time for the overall proof by only 2.7%.

One should not get the idea that the linear procedure is not doing anything for us. As we have already said, the presence of built-in arithmetic speeds up the theorem prover dramatically and makes the system far more rugged when applied to arithmetic problems. But the timing difference between the simple algorithm and theoretically more efficient ones is insignificant. Furthermore, it is not necessarily the case that a more efficient propagation algorithm would make the interface run faster. In particular, if the faster algorithm used a more complicated data structure for the data base and required a destructive push operation, it is probably the case that the interface would spend more time than it does now in such activities as popping the data base and exploring it for the key multiplicands.

## 10. CONCLUSION

We have made and documented three observations: linear arithmetic is inadequate for the arithmetic needs of program verification; integrating a linear arithmetic procedure into a theorem prover for a richer theory is surprisingly difficult; and the theoretical efficiency of a linear arithmetic procedure is a poor measure of its utility to a larger system.

We believe these same observations can be made about decision procedures in general. Let us quickly review our observations while considering decision procedures.

Decidable theories are inadequate for the specification of most programs. The situation is improved somewhat by the work of Oppen and Nelson (1979) which shows how one can construct a system of co-operating decision procedures for disjoint theories. But in our experience most theories of use to program verification are not disjoint. For example, the function LEN connects the theory of lists to that of the naturals and DELTA1 connects character strings to naturals.

But what makes it hardest to apply the work on decision procedures to program verification is the presence of user defined functions. DELTA1 is one example of a function that cannot be anticipated by the designer of the decision procedure. Other examples we have seen recently are: 'the number of times *X* occurs in *Y*', 'the number of processors that voted for *X*', and 'the length of the non-circular path defined by tracing the non-0 indices stored in the array *A* starting at location *I*'. Such functions are introduced not by the designer of the theorem prover but by the user when he is confronted with the need to specify a given program. Since decision procedures for these extended theories are not generally available, one must have more powerful proof techniques or be forced to assume the more doubtful conjectures behind a program's correctness.

But decidable theories are common fragments of the theories used in the specification of programs. It is thus useful to integrate decision procedures with the more powerful methods. A natural goal is to make it unnecessary for the more powerful system to derive from explicit axioms and lemmas the theorems of the decidable theory. To achieve such integration is very difficult because one must identify each use to which the heuristic theorem prover puts axioms and lemmas and make the decision procedure serve in each of those roles.

Furthermore, the black box nature of the decision procedure is frequently destroyed by the need to integrate it. The integration forces into the theorem prover much knowledge of the inner workings of the procedure and forces into the procedure many features that are unnecessary when the problem is considered in isolation. Thus it is not possible to substitute one decision procedure for another nor can the selection (much less the implementation) of the original procedure be entirely independent of the needs of the larger system.

Finally, the time spent in the interface between the heuristic theorem prover and the decision procedure may dominate that spent in the decision procedure itself. Since efficiency in the decision procedure may not gain much overall, it is often not worth the effort to select more efficient procedures because of the complicated data structures and inflexible control strategies they employ to gain efficiency.

When sufficiently powerful theorem provers for program verification are finally produced they will undoubtedly contain many integrated decision procedures. But despite the fact that work on decision procedures is elegant, easily published, mathematically pleasing, and demands rather limited computational resources, the usefulness of that work to program verification is not easily evaluated. The difference between a black box and an integrated decision procedure is a lot of work. It is probably the case that much hard work on any given black box will be scrapped when the box is torn apart and reassembled inside a larger system. Indeed, we believe that the work on many procedures is simply irrelevant to the goal of constructing useful mechanical theorem provers since the use of a faster procedure will not necessarily speed up the overall system. We believe that the development of useful procedures for program verification must take into consideration the problems of connecting those procedures to more powerful theorem provers.

**Acknowledgments**

The research reported here was supported by National Science Foundation Grant MCS-8202943 and Office of Naval Research Contract N00014-81-K-0634.

## REFERENCES

Bledsoe, W. W. (1975) A new method for proving certain Presburger formulas. Advance Papers, *Fourth Int. Joint Conf. on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp. 15–20.

Bledsoe, W. W. and Hines, L. M. (1980) Variable elimination and chaining in a resolution-based prover for inequalities. In *5th Conference on Automated Deduction, Lecture Notes in Computer Science* (eds W. Bibel and R. Kowalski) pp. 70–87. Springer-Verlag, Berlin.

Boyer, R. S. and Moore, J. S. (1977) A fast string searching algorithm. *Commun. ACM* **20,** 762–772.

Boyer, R. S. and Moore, J. S. (1979) *A computational logic.* Academic Press, New York.

Boyer, R. S. and Moore, J. S. (1981a) Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The correctness problem in computer science* (eds R. S. Boyer and J. S. Moore). Academic Press, London.

Boyer, R. S. and Moore, J. S. (1981b) A verification condition generator for FORTRAN. In

*The correctness problem in computer science* (eds R. S. Boyer and J. S. Moore) Academic Press, London.

Boyer, R. S. and Moore, J. S. (1984) Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly* **91**, 181–189.

Cooper, D. C. (1972) Theorem proving in arithmetic without multiplication. In *Machine Intelligence* 7 (eds B. Meltzer and D. Michie), pp. 91–99. Edinburgh University Press, Edinburgh.

Gloess, P. Y. (1980) An experiment with the Boyer–Moore theorem prover: a proof of the correctness of a simpler parser of expressions. In *5th Conference on Automated Deduction, Lecture Notes in Computer Science,* pp. 154–169. Springer-Verlag, Berlin.

Hodes, L. (1971) Solving problems by formula manipulation. *Proc. Second Int. Joint Conf. on Artificial Intelligence,* 553–559. The British Computer Society.

King, J. C. (1969) A program verifier. Ph.D. Thesis, Carnegie–Mellon University.

Moore, J. S. (1979) A mechanical proof of the termination of Takeuchi's function. *Information Processing Letters* **9**, 176–181.

Nelson, G. and Oppen, D. C. (1979) Simplification by cooperating decision procedures. *ACM Transactions of Programming Languages* **1**, 245–257.

Russinoff, D. M. (1983) A mechanical proof of Wilson's theorem, Department of Computer Sciences, University of Texas at Austin.

Shankar, N. (1985) A mechanical proof of the Church–Rosser theorem. ICSCA-CMP-45, Institute for Computing Science, University of Texas at Austin.

Shostak, R. (1977) On the SUP-INF method for proving Presburger formulas. *JACM* **24**, 529–543.

Shostak, R. (1978) Deciding linear inequalities by computing loop residues, Computer Science Laboratory, SRI International, Menlo Park, Calif.

Shostak, R. (1979) A practical decision procedure for arithmetic with function symbols. *JACM* **26**, 351–360.

6

# A Problem Simplification Approach that Generates Heuristics for Constraint-Satisfaction Problems

R. Dechter and J. Pearl
Department of Computer Science,
University of California at Los Angeles, USA

**Abstract**
Many AI tasks can be formulated as constraint-satisfaction problems (CSPs), i.e. the assignment of values to variables subject to a set of constraints. Recognition of three-dimensional objects, puzzle solving, electronic circuit analysis and truth-maintenance systems are examples of such problems, and these are normally solved by various versions of backtrack search. In this work we show how advice can be automatically generated to guide the order in which the search algorithm assigns values to the variables, so as to reduce the amount of backtracking. The advice is generated by consulting relaxed models of the subproblems created by each value-assignment candidate. The relaxed problems are chosen to yield backtrack-free solutions, and the information retrieved from these models induces a preference order among the choices pending in the original problem.

We identify a class of CSPs whose syntactic and semantic properties make them easy to solve. The syntactic properties involve the structure of the constraint graph while the semantic properties guarantee some local consistencies among the constraints. In particular, tree-like constraint graphs can be easily solved and are chosen therefore as the target model for the relaxation scheme. Optimal algorithms for solving easy problems are presented and analysed. A scheme for constructing a 'best' constraint-tree approximation to a given constraint graph is introduced and, finally, the utility of using the advice is evaluated in a synthetic domain of CSP instances.

## 1. BACKGROUND AND MOTIVATION

### 1.1. Introduction

An important component of human problem-solving expertise is the ability to use knowledge about solving easy problems to guide the solution of difficult ones. Only a few works in AI (Sacerdoti, 1974; Carbonell 1983) have attempted to equip machines with similar capabilities. Gaschnig (1979), Guida and Somalvico (1979), and Pearl (1983) suggested that knowledge about easy problems could be instrumental in the mechanical discovery of heuristics. Accordingly, it should be possible to manipulate the representation of a difficult problem until it is approximated by an easy one, solve the easy problem, then use the solution to guide the search process in the original problem.

The implementation of this scheme requires three major steps: (a) simplification; (b) solution; and (c) advice generation. Additionally, to perform the simplification step, we must have a simple, *a priori* criterion for deciding when a problem lends itself to easy solution.

This paper uses the domain of constraint-satisfaction tasks to examine the feasibility of these three steps. It establishes criteria for recognizing classes of easy problems, provides special procedures for solving them, demonstrates a scheme for generating good relaxed models, and introduces an efficient method for extracting advice from them. Finally, the utility of using the advice is evaluated in a synthetic domain of problem instances.

Constraint-satisfaction problems (CSPS) involve the assignment of values to variables subject to a set of constraints. Understanding three-dimensional drawings, graph colouring, electronic circuit analysis, and truth-maintenance systems are examples of CSPS. These are normally solved by some version of backtrack search which may require exponential search time (for example, the graph-colouring problem is known to be NP-complete).

The following paragraphs summarize the basic terminology of the theory of CSP as presented in Montanari (1974) and extended by Mackworth (1977) and Freuder (1982). Some observations are presented regarding the relationships between the representation of the problem and the performance of the backtrack algorithm.

## 1.2. Definition and nomenclature

Formally, the underlying model of a CSP involves a set of $n$ variables $X_1, \ldots, X_n$ each having a set of domain values $D_1, \ldots, D_n$. An $n$-ary relation on these variables is a subset of the Cartesian product:

$$\rho \subseteq D_1 \times D_2 \times, \ldots, \times D_n. \tag{1}$$

A binary constraint $R_{ij}$ between two variables is a subset of the Cartesian product of their domain values, i.e.

$$R_{ij} \subseteq D_i \times D_j. \tag{2}$$

A network of binary constraints is the set of variables $X_1, \ldots, X_n$ plus the set of binary constraints between pairs of variables and it represents an $n$-ary relation defined by the set of $n$-tuples that satisfy all the constraints. Formally, given a symmetric network of constraints between

$n$ variables, the relation $\rho$ represented by it is:

$$\rho = \{(x_1, x_2, \ldots, x_n) \mid x_i \in D_i, \text{ and } (x_i, x_j) \in R_{ij} \text{ for all } i, j\}. \tag{3}$$

Not every $n$-ary relation can be represented by a network of binary constraints with $n$ variables, and the issues of finding the best approximation by such network are addressed in Montanari (1974). In this paper we will discuss only relations induced by a network of binary constraints and henceforth assume that all constraints are binary and symmetric.

Each network of constraints can be represented by a constraint graph where the variables are represented by nodes in the graph and the non-universal constraints by arcs. The constraints themselves can be represented by the set of pairs they allow, or by a matrix in which rows and columns correspond to values of the two variables and the entries are 0 or 1 depending on whether or not the corresponding pair of values is allowed by the constraint. Figure 1(a) displays a typical network of constraints, where constraints are given using matrix notation as in Figure 1(b).

Several operations on constraints can be defined. The useful ones are: union, intersection, and composition. The union of two constraints between two variables is a constraint that allows all pairs that are allowed by either one of them. The intersection of two constraints allows only pairs that are allowed by both constraints. The composition of two constraints, $R_{12}$, $R_{23}$ 'induces' a constraint $R_{13}$ defined as follows: a pair $(x_1, x_3)$ is allowed by $R_{13}$ if there is at least one value $x_2 \in D_2$ such that $(x_1, x_2) \in R_{12}$ and $(x_2, x_3) \in R_{23}$. If matrix notation is used to represent constraints, then the induced constraint $R_{13}$ can be obtained by matrix multiplication:

$$R_{13} = R_{12} \cdot R_{23}. \tag{4}$$

A partial order among the constraints can be defined as follows: $R_{ij} \subseteq R'_{ij}$ iff every pair allowed by $R_{ij}$ is also allowed by $R'_{ij}$ (this is exactly set inclusion). In this case we say that $R_{ij}$ is smaller than $R'_{ij}$. We can also say that $R'_{ij}$ is a relaxation of $R_{ij}$. The smallest constraint between variables $X_i$ and $X_j$ is the empty constraint, denoted $\Phi_{ij}$, which does not

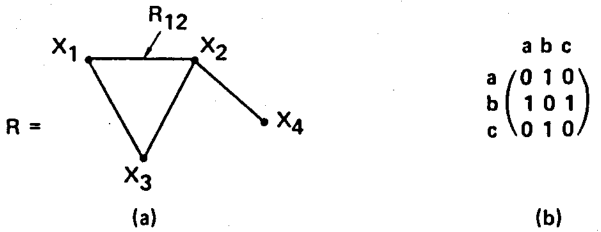


Figure 1.

allow any pair of values. The largest is the universal constraint, denoted $U_{ij}$, which permits all possible pairs. A corresponding partial order can be defined among network of constraints having the same set of variables. We say that $R \subseteq R'$ if the partial order is satisfied for every pair of corresponding constraints in the networks.

Finally, we define the notion of equivalence among networks of constraints: two networks of constraints with the same set of variables are equivalent if they represent the same $n$-ary relation.

Consider, for example, the network of Figure 2, representing a problem of four variables, each having the two-valued domain $\{1, 0\}$. The constraints are attached to the arcs and are given, in this case, by a set of pairs. The direction of the arcs only indicates the way by which constraints are specified. The constraint between $X_1$ and $X_4$, displayed in Figure 2(b), can be induced by $R_{12}$ and $R_{24}$. Therefore, adding this constraint to the network will result in an equivalent network. Similarly, since the constraint $R_{21}$ can be induced from $R_{23}$ and $R_{31}$ it can be deleted without changing the relation represented by the network.

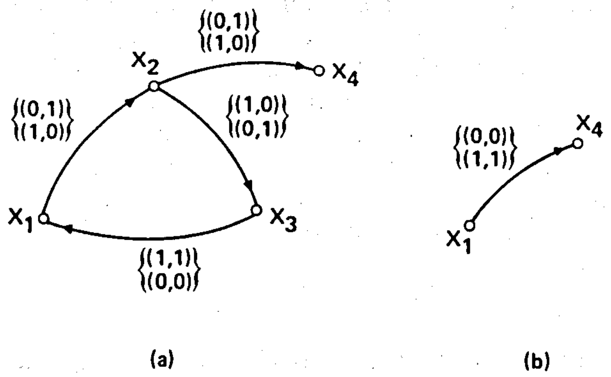


Figure 2.

The process of inducing relations in a given network makes the constraints smaller and smaller, while leaving the networks equivalent to each other. Montanari called the smallest network of constraints which is equivalent to a given network $R$ the 'Minimal Network'. The minimal network of constraints makes the 'global' constraints on the network as 'local' as possible. In other words, a minimal network of constraints is perfectly explicit.

Usually a CSP problem is described by a network of constraints. A tuple in the relation represented by the network is called a solution. The problem is either to find all solutions, one solution, or to verify that a certain tuple is a solution. The last problem is fairly easy while the first two problems can be difficult and have attracted a substantial amount of research.

### 1.3. Backtrack for CSP

The algorithm mostly used to solve CSP problems is Backtrack. Given a vertical order of the set of variables $X_1, X_2, \ldots, X_n$ and a horizontal order of values in each variable's domain $x_{i,1}, x_{i,2}, \ldots, x_{i,k}$, algorithm Backtrack for finding one solution is given below:

```
Backtrack
Begin
  1. Assign $x_{1,1}$ to $X_1$ (if allowed by a unary constraint)
  2. $k = 1$
  3. while $k \leq n - 1$
  4.    while $X_{k+1}$ has more values /* values $x_1, x_2, \ldots, x_k$ were
        already selected */
  5.       choose the first value $x_{k+1,j}$ of $X_{k+1}$, such that
           $(x_1, x_2, \ldots, x_k, x_{k+1,j})$ satisfies all constraints
  6.       then erase (temporarily) $x_{k+1,1}, \ldots, x_{k+1,j}$ from domain of
           $X_{k+1}$
  7.          $k = k + 1$
  8.          goto 3
  9.    end
 10.    $k = k - 1$ (backtrack since no value at (5) exists)
 11.    If $k = 0$ exit, no solution exists
 12. end
 13. exit with solution
End.
```

In line 5 of the algorithm all the constraints between $X_{k+1}$ and previous variables in the vertical order are checked. The value chosen should be consistent with all the previous instantiated values under those constraints. For Backtrack to find all solutions the above algorithm should be modified slightly by adding another outer loop and terminating only when $k = 0$.

Montanari considered the question of finding the minimal network $M$ of a given network $R$ as the central problem in CSPs implying that once it is available the problem is solved. The following two lemmas elaborate on this issue by relating the minimal network to the Backtrack algorithm.

*Lemma* 1. Let $R$ and $R'$ be two equivalent networks such that $R' \subseteq R$, then given the same order for instantiating variables, any sequence of values that is explicated by Backtrack with $R'$ will be explicated also by Backtrack with $R$ when Backtrack looks for all solutions.

*Proof.* The order between the networks implies that any sequence of values which is consistent under $R'$ is also consistent under $R$.

*Conclusions*: Given a network $R$ and a fixed order of variables' instantiation, Backtrack's performance, when looking for all solutions, is most efficient on the minimal network, relative to all networks which are equivalent to $R$, since it is contained in all of them.

We now show that when the algorithm seeks only one solution then, with the minimal network, the solution can be found easily in many cases. Some more definitions are required.

Given an $n$-ary relation $\rho$, representable by a network with $n$ variables, the projection $\rho_S$ of the relation $\rho$ on a subset $S$ of the variables is not always representable by a network with $|S|$ nodes. If for any subset of variables, $S$, $\rho_S$ is representable by a network with $|S|$ variables then $\rho$ is said to be a 'Decomposable relation'. Given an $n$-ary decomposable relation $\rho$, represented by a minimal network $M$, then for any subset $S$ of variables the subnetwork of $M$ restricted to the nodes in $S$, is a minimal network of $\rho_S$. In this case $M$ is also said to be decomposable.

For example, the network in Figure 3 is minimal but not decomposable. The relation represented by $M$ is:

$$\rho = \{(x_{1,1}, x_{2,1}, x_{3,1}, x_{4,1}), (x_{1,1}, x_{2,2}, x_{3,2}, x_{4,2}), (x_{1,2}, x_{2,2}, x_{3,1}, x_{4,3})\}. \tag{5}$$

(Note that $X_4$ is a non-binary variable.)

If $S = \{X_1, X_2, X_3\}$ it can be shown that

$$\rho_S = \{(x_{1,1}, x_{2,1}, x_{3,1}), (x_{1,1}, x_{2,2}, x_{3,2}), (x_{1,2}, x_{2,2}, x_{3,1})\} \tag{6}$$

cannot be represented by a network with three variables. (For more details see Montanari, 1974.)

*Lemma* 2. If $M$ is a minimal and decomposable network then Backtrack will find one solution without backtracking at all.

*Proof.* From $M$'s decomposability it follows that any projection $\rho_S$ has a minimal network which is the subnetwork of $M$ that is restricted to the variables in $S$. Therefore any tuple of a subset of the variables $S$, that satisfy all the constraints in the minimal subnetwork is part of an $n$-tuple in the $n$-ary relation represented by $M$, and therefore it can always be extended.

The complexity of finding a solution given a minimal and decomposable network $M$, is, therefore, $O(n^2k)$ when $n$ is the number of

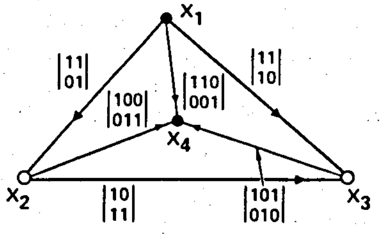


Figure 3.

variables and $k$ is the maximum cardinality of the value domain for all variables. In the previous example of a non-decomposable minimal network Backtrack may explore the path $x_{1,1}, x_{2,2}, x_{3,1}$ and since it cannot be extended to a 4-tuple relation satisfying $M$ the algorithm will have to backtrack. In conclusion we see that when solving a CSP, finding all or one solution is much easier when the minimal network is available. It is still not clear that it is always easy.

Backtracking and its performance on CSPs have been extensively discussed in the AI literature. Most researchers are trying to identify the major maladies in its performance, to provide a corresponding cure, and to analyse the results. The discussion can be separated along the following lines:

1. The problem objectives: finding all or finding one solution
2. Control parameters: controlling the order of variables' instantiation, order of values' instantiation, or manipulating the problem's representation by pruning values or propagating constraints.
3. Cure implementation: performing the cures themselves prior to the start of the algorithm, as a pre-processing phase, or incorporating them dynamically into the algorithm while it searches for solution(s).

Mentioning only a few studies, we start with Montanari (1974) who considered the problem of finding all solutions and discussed the solution of a problem by propagating the constraints and pruning pairs of values from them. In light of the previous lemmas his work can be regarded as a pre-processing phase to a backtrack algorithm. Mackworth (1977) extended Montanari's work by considering explicitly Backtrack and trying to cure its maladies by essentially the same approach, namely, pruning certain values from a variable's domains altogether. Haralick and Elliot (1980) discussed the problem of finding all solutions and examined various methods of pruning values. They suggested possible lookahead mechanisms which are incorporated into the algorithm. Freuder (1982) considered the problem of finding one solution to a CSP and provided a procedure to select a good ordering of variables which is performed in a pre-processing phase of Backtrack. Other works in analysing the average performance of Backtrack were reported by Nudel (1983), Purdom and Brown (1985), and Haralick and Elliot (1980) all estimating the size of the tree exposed by Backtrack while searching for all solutions.

It seems that the only parameter not considered for controlling Backtrack's performance is the order in which values are assigned to variables. Part of the reason can be explained by the following theorem.

*Theorem* 1. Given the objective of finding all solutions and given a fixed vertical order for the instantiation of variables, the search tree exposed by Backtrack is invariant to the order of the selection of values. (All search trees which are identical up to an ordering of branches are considered the same.)

*Proof.* Any sequence of values that is explored by Backtrack with respect to a specific order of variables is consistent under this subset of variables and it may or may not lead to a solution. The only way Backtrack can find out if it is extendable to a solution is by continuing to explore it. Therefore, Backtrack, which tries to find all solutions, will have to search this sequence for all orders of assigning values.

Similarly, Backtrack when looking for one solution, in a CSP that has no solution, will expose the same search tree under any order of value assignment, given a fixed vertical order.

The above theorem explains why value-selection strategies were not devised to improve Backtrack's performance for the case of all solutions. In this paper we address the objective of finding a single solution to CSPs. Although this problem is easier it can still be very difficult (e.g. three-colourability) and it appears frequently. Theorem proving, planning and even vision problems are examples of domains where finding one solution will normally suffice. For such problems, the order by which values are selected may have a profound effect on the algorithm's performance. In the following sections we outline an approach to devise value selection strategies.

### 1.4. General approach for automatic advice generation

In this section we show how the approach of solving difficult problems by consulting easier versions assists the solution of CSPs by Backtrack. The approach will consist of the three steps, mentioned in the Introduction, i.e. simplification, solution, and advice generation. Following the model of the $A^*$ algorithm that uses heuristics to guide the selection of the next node of expansion, we now wish to guide Backtrack in selecting the next node on its path. We assume that the order of variables is fixed and therefore the selection of the next node amounts to choosing a promising value from a set of pending options. Clearly, if the next value can be guessed correctly, and if a solution exists, the problem will be solved in linear time with no backtracking. Backtrack builds partial solutions and extends them as long as it believes that they are part of a whole solution. When a deadend is recognized it backtracks to a previous variable. The advice we wish to generate should order the candidates according to the confidence we have that they can be extended further to a solution.

Such confidence can be obtained by making simplifying assumptions about the continuing portion of the search graph and estimating the likelihood that it will contain a solution even when the simplifying assumptions are removed. It is reasonable to assume that if the simplifying assumptions are not too severe then the number of solutions found in the simplified version of the problem would correlate positively with the number of solutions present in the original version. We therefore, propose to count the number of solutions in the simplified model and use it as a measure of confidence that options considered will lead to an overall solution.

To incorporate the advice generation into the Backtrack algorithm, line 5 should be exchanged with the following:

```
5a.  eliminate all values of $X_{k+1}$ which are not consistent with
     $x_1, \ldots, x_k$
5b.  /*let $x_{k+1,1}, \ldots, x_{k+1,t}$ all the remaining candidates for
     assignment*
     /advise $((x_{k+1,1}, \ldots, x_{k+1,t}), (x'_{k+1,1}, \ldots, x'_{k+1,t}))$
5c.  assign $x'_{k+1,1}$ to $X_{k+1}$
```

The advise procedure takes the set of consistent values of $X_{k+1}$ and order them according to the estimates of the number of possible solutions stemming from them.

The essence of the remaining sections is to describe the advice-giving algorithm and provide theoretical grounds for it. In Section 2 we establish criteria for recognizing classes of easy CSPs and introduce an efficient method of counting the number of solutions. In Section 3 the process of simplification of any CSP to an easy relaxed one that is also 'close' to it is addressed and in Section 4 the algorithm is implemented and the utility of using the advice is evaluated in a synthetic domain of CSPs.

## 2. THE ANATOMY OF EASY CSPs

### 2.1. Introduction and background

In general, a problem is considered easy when its representation permits a solution in polynomial time. However, since we are dealing mainly with backtrack algorithms, we will consider a CSP easy if it can be solved by a backtrack-free procedure. A backtrack-free search is one in which Backtrack completes without backtracking, thus producing a solution in time linear with the number of variables.

The discussion of backtrack-free CSPs relies heavily on the concept of constraint graphs. Freuder (1982) has identified sufficient conditions for a constraint graph to yield a backtrack-free solution, and has shown, for example, that tree-like constraint graphs can be made to satisfy these conditions, with a small amount of pre-processing. Our main purpose here is to study further classes of constraint graphs lending themselves to backtrack-free solutions and to devise efficient algorithms for solving them. Once these classes are identified they can be chosen as targets for a problem simplification scheme: constraints can be selectively deleted from the original specification so as to transform the original problem into a backtrack-free one. As already mentioned, we propose to use the 'number of consistent solutions in the simplified problem' as a figure-of-merit to establish priority of value assignments in the backtracking search of the original problem. We show that this figure of merit can be computed in a time comparable to that of finding a single solution to an easy problem.

*Definition. An ordered constraint graph* is a constraint graph in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by the backtrack search algorithm. The *width of a node* is the number of arcs that lead from that node to previous nodes, the *width of an ordering* is the maximum width of all nodes, and the *width of a graph* is the minimum width of all the orderings of that graph (Freuder, 1982).

Figure 4 presents six possible orderings of a constraint graph. The width of node $C$ in the first ordering (from the left) is 2, while in the second ordering it is 1. The width of the first ordering is 2, while that of the second is 1. The width of the constraint graph is, therefore, 1. Freuder provided an efficient algorithm for finding both the width of a graph and the ordering corresponding to this width. He further showed that a constraint graph is a tree iff it is of width 1.

Montanari (1974) and Mackworth and Freuder (1977) have introduced two kinds of local consistencies among constraints: *arc consistency* and *path consistency*. Their definitions assume that the graph is directed, i.e. each symmetric constraint is represented by two directed arcs.

Let $R_{ij}(x, y)$ stand for the assertion that $(x, y)$ is permitted by the explicit constraint $R_{ij}$.

*Definition.* Directed arc $(X_i, X_j)$ is *arc consistent* iff for any value $x \in D_i$ there is a value $y \in D_j$ such that $R_{ij}(x, y)$ (Mackworth, 1977).

*Definition.* A path of length $m$ through nodes $(i_0, i_1, \ldots, i_m)$ is *path consistent* if for any value $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $R_{i_0 i_m}(x, y)$, there is a sequence of values $z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that

$$R_{i_0 i_1}(x, z_1) \text{ and } R_{i_1 i_2}(z_1, z_2) \text{ and } \cdots R_{i_{m-1} i_m}(z_{m-1}, y). \tag{7}$$

$R_{i_0 i_m}$ may also be the universal relation, e.g. permitting all possible pairs (Montanari, 1974).

A constraint graph is arc (path) consistent if each of its directed arcs (pa:h) is arc (path) consistent. Achieving 'arc consistency' means deleting certain values from the domains of certain variables such that the resultant graph will be arc consistent, while still representing the same overall set of solutions. To achieve path consistency, certain pairs of values that were initially allowed by the local constraints should be

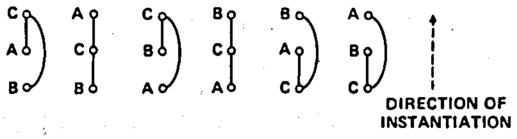


Figure 4.

disallowed. Montanari and Mackworth have proposed polynomial-time algorithms for achieving arc consistency and path consistency. In Mackworth and Freuder (1984) it is shown that arc consistency can be achieved in $o(ek^3)$ while path consistency can be achieved in $o(n^3k^5)$, where $n$ is the number of variables, $k$ is the number of possible values, and $e$ is the number of edges.

The following theorem is due to Freuder.

*Theorem* 2.

(a) If the constraint graph has width 1 (i.e. the constraint graph is a tree) and if it is arc consistent then it admits backtrack-free solutions.

(b) If the width of the constraint graph is 2 and it is also path consistent then it admits backtrack-free solutions.

The above theorem suggests that tree-like CSPS (CSPS whose constraint graph are trees) can be solved by first achieving arc consistency and then instantiating the variables in an order which makes the graph have width 1. Since this backtrack-free instantiation takes $O(ek)$ steps, and on trees $O(nk)$, the whole problem can be solved in $O(nk^3)$ and therefore tree-like CSPS are easy. The test for this property is also easily verified: to check whether or not a given graph is a tree can be done by a regular $O(n^2)$ spanning tree algorithm.

The second part of the theorem tempts us to conclude that a width-2 constraint graph should admit a backtrack-free solution after passing through a path-consistency algorithm. In this case, however, the path-consistency algorithm may add arcs to the graph and increase its width beyond 2. This often happens when the algorithm deletes value-pairs from a pair of variables that were initially related by the universal constraint (having no connecting arc between them), and it is often the case that passage through a path-consistency algorithm renders the constraint graph complete. It may happen, therefore, that no advantage could be taken of the fact that a CSP possesses a width-2 constraint graph if it is not already path consistent. We are not even sure whether width-2 suffices to preclude NP-completeness.

In the following section we give weaker definitions of arc and path consistency which are also sufficient for guaranteeing backtrack-free solutions but have two advantages over those defined by Montanari (1974) and Mackworth (1977):

(a) they can be achieved more efficiently; and

(b) they add fewer arcs to the constraint graph, thus preserving the graph width in a larger class of problems.

## 2.2. Algorithms for achieving directional consistency

*The case of Width*-1. In constraint graphs which are trees, full arc consistency is more than what is actually required for enabling backtrack-free solutions. For example, if the constraint graph in Figure 5 is ordered

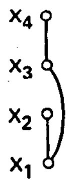


Figure 5.

by $(X_1, X_2, X_3, X_4)$, nothing is gained by making the directed arc $(X_3, X_1)$ consistent.

To ensure backtrack-free assignment, we need only make sure that any value assigned to variable $X_1$ will have at least one consistent value in $D_3$. This can be achieved by making only the directed arc $(X_1, X_3)$ consistent, regardless of whether $(X_3, X_1)$ is consistent or not. We, therefore, see that arc consistency is required only with respect to a single direction, the one specified by the order in which Backtrack will later choose variables for instantiations. This motivates the following definitions.

*Definition.* Given an order $d$ on the constraint graph $R$, we say that $R$ is $d$-arc-consistent if all the directed edges which follow the order $d$ are arc consistent.

*Theorem* 2. Let $d$ be a width-1 order of a constraint tree $T$. If $T$ is $d$-arc-consistent then the backtrack search along the order $d$ is backtrack-free.

*Proof.* Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The variable $X_{k+1}$ is connected to at most one previous variable (follows from the width-1 property), say $X_i$, which was assigned the value $x_i$. Since the directed arc $(X_i, X_{k+1})$ is along the order $d$, its arc consistency implies the existence of a value $x_{k+1}$ such that the pair $(x_i, x_{k+1})$ is permitted by the constraint $R_{i(k+1)}$. Thus, the assignment of $x_{k+1}$ is consistent with all previous assignments.

An algorithm for achieving directional arc consistency for any ordered constraint graph is given next (The order $d = (X_1, X_2, \ldots, X_n)$ is assumed.)

DAC(*D-ARC-CONSISTENCY*)

```
1. begin
2.    For $i = n$ to 1 by $-1$ do
3.       For each arc $(X_j, X_i)$; $j < i$ do
4.          REVISE$(X_j, X_i)$
5.       end
6.    end
7. end.
```

The algorithm REVISE$(X_j, X_i)$, given in Mackworth (1977), deletes values from the domain $D_j$ until the directed arc $(X_j, X_i)$ is arc consistent.

```
REVISE($X_j$, $X_i$)
  1. begin
  2.    For each $x \in D_j$ do
  3.       if there is no value $y \in D_i$ such that $R_{ji}(x, y)$ then
  4.       delete $x$ from $D_j$
  5.     end
  6. end.
```

To prove that the algorithm achieves $d$-arc-consistency we have to show that upon termination, any arc $(X_j, X_i)$ along $d(j<i)$, is arc consistent. The algorithm revises each $d$-directed arc once. It remains to be shown that the consistency of an already processed arc is not violated by the processing of coming arcs. Let arc $(X_j, X_i)$ $(j<i)$ be an arc just processed by REVISE$(X_j, X_i)$. To destroy the consistency of $(X_j, X_i)$ some values should be deleted from the domain of $X_i$ during the continuation of the algorithm. However, according to the order by which REVISE is performed from this point on, only lower indexed variables may have their set of values updated. Therefore, once a directed arc is made arc-consistent its consistency will not be violated.

The algorithm AC-3 (Mackworth, 1977) that achieves full arc-consistency is given for reference:

```
AC-3
  1. begin
  2.    $Q < -\{X_i, X_j) \mid (X_i, X_j) \in$ arcs, $i \neq j\}$
  3.      while $Q$ is not empty do
             select and delete arc $(X_k, X_m)$ from $Q$
  5.         REVISE$(X_k, X_m)$
  6.         if REVISE$(X_k, X_m)$ caused any change then
  7.            $Q < -Q \cup \{(X_i, X_k) \mid (X_i, X_k) \in$ arcs, $i \neq k, m\}$
  7.      end
  8. end.
```

The complexity of AC-3, achieving full arc consistency, is $O(ek^3)$. By comparison, the directional arc-consistency algorithm takes $ek^2$ steps since the REVISE algorithm, taking $k^2$ tests, is applied to every arc exactly once. It is also optimal, because even to verify directional arc-consistency each arc should be inspected once, and that takes $k^2$ tests. Note that when the constraint graph is a tree, the complexity of the directional arc-consistency algorithm is $O(nk^2)$.

*Theorem* 4. A tree-like CSP can be solved in $O(nk^2)$ steps and this is optimal.

*Proof.* Given that we know that the constraint graph is a tree, finding an order that will render it of width-1 takes $O(n)$ steps. A width-1 tree-CSP can be made $d$-arc-consistent in $O(nk^2)$ steps, using the DAC algorithm. The backtrack-free solution on the resultant tree is found in $O(nk)$. Finding a solution to tree-like CSPs takes, therefore, $O(nk) + O(nk^2) +$

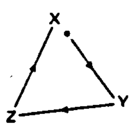
$O(n) = O(nk^2)$. This complexity is also optimal since any algorithm for solving a tree-like problem must examine each constraint at least once, and each such examination may take $k^2$ steps in the worst case (especially when no solution exists and the constraints permit very few pairs of values).

Interestingly, if we apply DAC with respect to order $d$ and then DAC with respect to the reverse order we get a full arc consistency for trees. We can, therefore, achieve full arc consistency on trees in $O(nk^2)$. Algorithm AC-3, on the other hand, can be shown to have a worst case performance on trees of $O(nk^3)$. Given, however, that the basic operation on constraints is REVISE, we shall next show that (full) arc consistency on general graphs cannot be achieved in less than $ek^3$ steps.

*Theorem* 5. A lower bound for achieving (full) arc consistency on graphs, using REVISE as the basic operation, is $\Omega(ek^3)$.

*Proof.* We present a problem instance that cannot be made arc consistent in less than $ek^3$. The problem, given in Figure 6, has $n$ variables (in the figure just three) connected in a cycle. We will describe only the three-element network. The example can be easily extended to any number of variables. Variable $X$ has $k$ values, variables $Y$, and $Z$, have $k+1$ values each. The constraint from $X$ to $Y$ maps values in $X$ to values in $Y$ which are incremented by 1. The constraints between $Y$ and $Z$ and between $Z$ and $X$ are both the equality mapping, except that $k+1$ of $Z$ is mapped to $k$ of $X$. The inconsistent arc is $(Y, X)$ since the value 0 of $Y$ has no pair in $X$. Removing 0 from $D_Y$ makes the arc $(Z, Y)$ inconsistent. This arc is examined and 0 is deleted, which make the arc $(X, Z)$ inconsistent, and so on. Since we assume that any examination of an arc is an $O(k^2)$ operation, and since only one value is deleted from an arc while it is examined, each arc will be examined $k$ times (there is no solution), and the complexity in this case is $\Omega(nk^3)$.

Returning to our primary aim of studying easy problems, we now show how advice can be generated for solving a difficult CSP using a relaxed tree-like approximation. Suppose that we want to solve an $n$-variables CSP using Backtrack with $X_1, X_2, \ldots, X_n$ as the order of instantiation. Let $X_i$ be the variable to instantiate next, with $x_{i1}, x_{i2}, \ldots, x_{ik}$ the possible candidate values. To minimize backtracking we should first try values



(a)

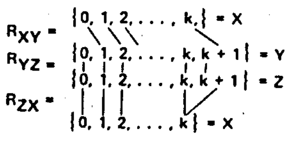

R_XY =
R_YZ =
R_ZX =
{0, 1, 2, . . . , k, } = X
{0, 1, 2, . . . , k, k + 1} = Y
{0, 1, 2, . . . , k, k + 1} = Z
{0, 1, 2, . . . , k} = X
(b)

Figure 6.

which are likely to lead to a consistent solution but, since this likelihood is not known in advance, we may estimate it by counting the number of consistent solutions that each candidate admits in some relaxed problem. We generate a relaxed tree-like problem by deleting some of the explicit constraints given, then count the number of consistent solutions containing each of the possible $k$ assignments, and finally use these counts as a figure of merit for scheduling the various assignments. In the following we show how the counting of consistent solutions can be imbedded within the $d$-arc-consistency algorithm, DAC, on trees.

Any width-1 order, $d$, on a constraint tree determines a directed tree in which a parent always precedes its children in $d$ (arcs are directed from the parent to its children). Let $N(x_{jt})$ stand for the number of solutions in the subtree rooted at $X_j$, consistent with the assignment of $x_{jt}$ to $X_j$. It can be shown that $N(\cdot)$ satisfy the following recurrence:

$$N(x_{jt}) = \prod_{\{c \mid X_c \text{ is a child of } X_j\}} \sum_{\{x_{cl} \in D_c \mid R_{jc}(x_{jt}, x_{cl})\}} N(x_{cl}). \tag{8}$$

From this recurrence it is clear that the computation of $N(x_{it})$ may follow the exact same steps as in DAC; simultaneously with testing that a given value $x_{jt}$, is consistent with each of its children nodes, we simply transfer from each child of $X_j$ to $x_{jt}$ the sum total of the counts computed for the child's values that are consistent with $x_{jt}$. The overall value of $N(x_{jt})$ will be computed later on by multiplying together the summations obtained from each of the children. Thus, counting the number of solutions in a tree with $n$ variables takes $O(nk^2)$, the same as establishing directional arc consistency. Recently Mohr and Henderson (1986) have reported an $(Olk^2)$ algorithm using a more elaborate book-keeping data structure.

*The case of width*-2. Order information can also facilitate backtrack-free search on width-2 problems by making path-consistency algorithms directional.

Montanari had shown that if a network of constraints is consistent with respect to all paths of length 2 (in the complete network) then it is path consistent. Similarly we will show that directional path consistency with respect to length-2 paths is sufficient to obtain a backtrack-free search on a width-2 problems.

*Definition.* A constraint graph, $R$, ordered with respect to $d = (X_1, X_2, \ldots, X_n)$, is *d-path consistent* if for every pair of values $(x, y)$, $x \in X_i$ and $y \in X_j$ such that $R_{ij}(x, y)$ and $i < j$, there exists a value $z \in X_k$, $k > j$ such that $R_{ik}(x, z)$ and $R_{kj}(z, y)$ for every $k > i, j$.

*Theorem* 6. Let $d$ be a width-2 order of a constraint graph. If $R$ is directional arc consistent and path consistent with respect to $d$ then it is backtrack-free.

*Proof.* To ensure that a width-2 ordered constraint graph will be backtrack-free it is required that the next variable to be instantiated will have values that are consistent with previous chosen values. Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The variable $X_{k+1}$ is connected to at most two previous variables (follows from the width-2 property). If it is connected to $X_i$ and $X_j$, $i, j < k$ then directional path consistency implies that for any assignment of values to $X_i$, $X_j$ there exists a consistent assignment for $X_{k+1}$. If $X_{k+1}$ is connected to one previous variable, then directional arc consistency ensures the existence of a consistent assignment.

An algorithm for achieving directional path consistency on any ordered graph will have to manage not only the changes made to the constraints but also the changes made to the graph, i.e. the arcs which are added to it. To describe the algorithm we use the matrix representation for constraints. The matrix $R_{ii}$ whose off-diagonal values are 0, represents the set of values permitted for variable $X_i$. The algorithm is described using the operations of intersection and composition. The intersection $R_{ij}$ of $R'_{ij}$ and $R''_{ij}$ is written: $R_{ij} = R'_{ij} \ \& \ R''_{ij}$.

Given a network of constraints $R = (V, E)$ and an order $d = (X_1, X_2, \ldots, X_m)$, we next describe an algorithm which achieves path consistency with respect to this order.

DCP-*d-path-consistency*

```
1. begin
2. $Y^0 = R$
3. for $k = n$ to 1 by $-1$ do
      (a) $\forall i \leq k$ connected to $k$ do
          $Y'_{ii} = Y^0_{ii} \ \& \ Y_{ik} \cdot Y_{kk} \cdot Y_{ki}$/* this is REVISE($i$, $k$)
      ($b$) $\forall i, j \leq k$ such that $(X_i, X_k), (X_j, X_k) \in E$ do
          $Y_{ij} = Y_{ij} \ \& \ Y_{ik} \cdot Y_{kk} \cdot Y_{kj}$
4.        $E <- E \cup (X_i, X_j)$
5.    end
6. end.
```

Step 3(a) is the equivalent of the REVISE($i$, $k$) procedure, and it performs the directional arc consistency. Step 3(b) updates the constraints between pairs of variables transmitted by a third variable which is higher in the order $d$. If $X_i$, $X_j$, $i$, $j < k$ are not connected to $X_k$ then the relation between the first two variables is not affected by $X_k$ at all. If only one variable, $X_i$, is connected to $X_k$, the effect of $X_k$ on the constraint $(X_i, X_j)$ will be computed by step 3(a) of the algorithm. The only time a variable $X_k$ affects the constraints between pairs of earlier variables is when it is connected to both. It is in this case only that a new arc may be added to the graph.

The complexity of the DCP algorithm is $O(n^3k^3)$. For variable $X_i$ the number of times the inner loop, 3(b), is executed is at most $O((i-1)^2)$ (the number of different pairs less than $i$), and each step is of order $k^3$. The computation of loop 3(a) is completely dominated by the computa-

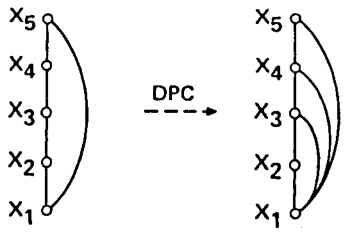


Figure 7.

tion of 3(b), and can be ignored. Therefore, the overall complexity is

$$\sum_{i=2}^{n} (i-1)^2 k^3 = O(n^3 k^3) \tag{9}$$

Applying directional path consistency to a width-2 graph may increase its width and therefore, does not guarantee backtrack-free solutions. Consequently it is useful to define the following subclass of width-2 CSP problems.

*Definition.* A constraint graph is *regular width-2* if there exists a width-2 ordering of the graph which remains width-2 after applying *d*-path-consistency, DPC.

A ring constitutes an example of a regular-width-2. Figure 7 shows an ordering of a ring's nodes and the graph resulting from applying the DPC algorithm to the ring. Both graphs are of width-2.

*Theorem* 7. A regular width-2 CSP can be solved in $O(n^3k^3)$.

*Proof.* Regular width-2 problem can be solved by first applying the DPC algorithm and then performing a backtrack-free search on the resulting graph. The first takes $O(n^3k^3)$ steps and the second $O(ek)$ steps.

A nice feature of regular width-2 CSPs is that they can be easily recognized and therefore can also be used as targets for simplification. Arnborg (1985) describes a linear time algorithm for recognizing width-2 graphs and generating the corresponding ordering (see also Bertele, 1972). For example, a tree of simple rings is easily recognizable as regular width-2 (see Figure 8).

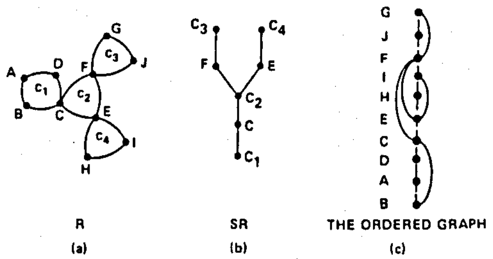


Figure 8.

### 2.3. Summary and conclusions

Of the three main steps involved in the process of generating advice—simplification, solution, and advice generation—we concentrated in this section on the following:

1. The simplification part: we have devised criteria for recognizing easy problems based on their underlying constraint graphs. The introduction of directionality into the notions of arc and path consistency enabled us to extend the class of recognizable easy problems beyond trees, to include regular width-2 problems.

2. The solution part: using directionality we were able to devise improved algorithms for solving simplified problems and to demonstrate their optimality. In particular, it is shown that tree-structured problems can be solved in $O(nk^2)$ steps, and regular width-2 problems in $O(n^3k^3)$ steps.

3. The advice generation part: we have demonstrated a simple method of extracting advice from easy problems to help Backtrack decide between pending options of value assignments. The method involves approximating the remaining part of a constraint-satisfaction task by a tree-structure problem, and counting the number of solutions consistent with each pending assignment. These counts can be obtained efficiently and can be used as figures-of-merit to rate the promise offered by each option.

## 3. THE SIMPLIFICATION PROCESS

The previous section suggests that a tree constraint-graph, being associated with an easy CSP, can be made a target to the simplification process from which advice will be extracted. We therefore discuss here the issues involved in approximating a network of binary constraints by a tree of constraints. We seek a good approximation since the closeness of the approximation tree to the original network will determine the reliability of the advice generated.

If the network $R$ has an equivalent tree representation we would obviously like to recognize it and find such a representation. This, however may not be explicit in the constraint network; a network may contain many redundant constraints which, if eliminated, still represent the same overall relation. For example, any one of the arcs in the network of Figure 9 can be eliminated producing a tree-structured

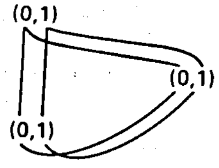


Figure 9.

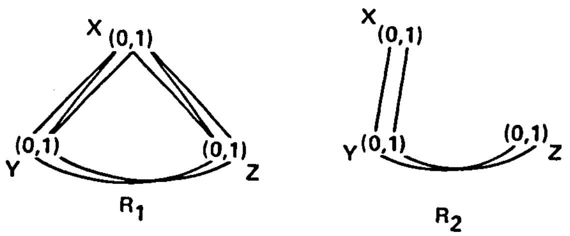


Figure 10.

constraint graph representing the same relation. Note that in this figure, and throughout this section, there are multiple arcs between variables which connect values. Two values are connected if they are permitted by the constraint. Another example is given in Figure 10 in which two three-node networks, $R_1$ and $R_2$, are displayed. These two networks are equivalent, because they both represent the equality relation $\rho = \{(0,0,0), (1,1,1)\}$ and, unlike that of Figure 9, both are maximal, i.e. the addition of any pair of values to any one of the constraints (relaxing any specific constraint) will result in a network representing a larger relation. Nevertheless, $R_1$ can be transformed into $R_2$ by simultaneously allowing the pair of values $(1,0)$ between $(Z, X)$ and disallowing the pair $(0,1)$ between $(X, Y)$. The question raised by this example is: what networks have a tree representation and how is the transformation into a tree to be performed.

The two examples given display two levels of operation to be considered in the process of transforming a network into a tree. The first is a macro operation involving the deletion of whole arcs (i.e. total elimination of constraints between a pair of variables) while the second micro operation, merely modifies the arcs by addition and deleting pairs of values. In our approach we will consider only macro operations of arc deletions; the use of micro transformations introduces a higher level of difficulty to which we will not relate at this point. Considering only arc deletions, a network $R$ can be transformed into an equivalent tree only if some of the arcs are redundant, i.e. they represent constraints that can be inferred from others. This immediately raises the question of testing whether a given constraint is implied by others.

The question is the inverse of that posed by Montanari (1974) who claimed that the central problem in CSPs is the transformation of the original network $R$ into its minimal representation, $M$, which is the most *redundant* network that represents the same relation as $R$. Our interest here is the opposite, transforming $R$ into one of its least explicit equivalent networks.

*Definition*

1. A network $R$ is maximal if there is no network $R'$ on the same domains, such that $R \subseteq R'$ and $R \sim R'$.

2. A network $R$ is *arc maximal* if any arc deletion results in a network representing a larger relation.

A maximal network is arc maximal but not necessarily vice versa.

*Lemma* 3. An arc consistent constraint tree is maximal.

*Proof.* In an arc consistent tree, for any permitted pair of values there is an $n$-tuple in the relation which contains this pair. Disallowing this pair will eliminate such a tuple from the relation, thus making the relation smaller. In other words any arc consistent constraint tree is a minimal network for that relation.

An immediate conclusion is that an arc-consistent tree network is arc maximal. In general a deletion of an arc from a tree constraint may result in a larger relation even when it is not arc consistent. Let $T_1$ and $T_2$ be the two disconnected subtrees generated from deleting arc $(A, B)$ and let $\rho_1$ and $\rho_2$ be the projection of $\rho$ on the variables in $T_1$ and $T_2$, respectively. The relation obtained after deleting the arc $(A, B)$ from $T$ is the product of $\rho_1$ and $\rho_2$ (i.e. any $n$-tuple that is the concatenation of a tuple in $\rho_1$ and a tuple in $\rho_2$. Therefore if there is a tuple in $\rho_1$ with $A = a$ and a tuple in $\rho_2$ with $B = b$ then the relation resulting from deleting arc $(A, B)$ permits the pair $(a, b)$.

In most cases a CSP will not be arc redundant, because if it is posed by humans its specification has already passed through some process of redundancy filtering, and therefore arc deletion will almost always generate larger relations. The third question on which we will focus, therefore, is: given a network of constraints, $R$, what is the spanning tree, $T$, that will best approximate $R$?

To discuss the quality of approximations, the notion of closeness of relations must be first agreed on. Let $\rho$ be the relation represented by $R$ and $\rho_a$ the relation represented by a relaxed network $R_a$. $\rho \subseteq \rho_a$. An intuitively appealing measure for the closeness of $R$ to $R_a$ may be:

$$M(R, R_a) = \frac{|\rho|}{|\rho_a|} \tag{10}$$

where $|\rho|$ is the number of $n$-tuples in $\rho$. This measure satisfies:

(a) $M(\rho, \rho) = 1$;

(b) if $\rho \subseteq \rho_a \subseteq \rho_b$ then $1 \geq M(R, R_a) \geq M(R, R_b)$.

$M$ is a global property of two relations and the task of finding the spanning tree which yields the lowest $M$ is very complex. Instead we propose a greedy approach: at each step the least 'valuable' arc, which leaves the network connected, is deleted, namely, the arc deleted that keeps the resulting network closest to the original one. To pursue this approach we need to define a measure of constraint strength, called *weight*, for each arc, that will estimate the contribution of that arc to the overall relation. Let $R$ be a network of constraints and $R'$ be the network after the arc $(X, Y)$ was eliminated, i.e. the constraint between $X$ and $Y$ becomes the universal constraint. Let $l$ and $l'$ be the size of the relation represented by $R$ and $R'$, respectively. $n'(x_i, y_j)$ is the number of tuples in the relation represented by $R'$ having $X = x_i$ $Y = y_j$, $R'(X, Y)$ is the constraint induced by $R'$ on the pair $(X, Y)$, $r(X, Y)$ is the local constraint given between $X$ and $Y$ in $R$.

The following is satisfied

$$l = l' - \sum_{(x_i, y_j) \in R(X, Y) - r(X, Y)} n'(x_i, y_i) \tag{11}$$

therefore

$$\frac{l}{l'} = 1 - \sum_{(x_i, y_j) \in R(X, Y) - r(X, Y)} n'(x_i, y_j)/l'. \tag{12}$$

Since we have no way of knowing the quantities $n'(x, y)$ and the structure of the induced constraint $R'(X, Y)$, we will estimate them both by a constant, $c$, and $R'$, respectively. We get:

$$\frac{l}{l'} = 1 - \frac{c}{l'} |R' - r(X, Y)|. \tag{13}$$

The only quantity we can actually examine is $R(X, Y)$; therefore to maximize $l/l'$ the above formula suggests choosing the constraint $r(X, Y)$ with the most number of allowed value pairs. Our first measure of constraint weight is, therefore, defined by:

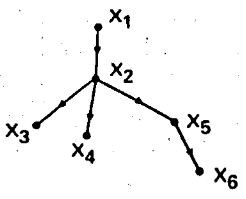
$$m_1(X, Y) = |r(X, Y)|. \tag{14}$$

For instance, the weight of the universal constraint is $m_1(X, Y) = k^2$, and if $r(X, Y) = \Phi$ then the weight becomes $m_1(X, Y) = 0$.

In what follows we develop another measure of constraint strength by adopting notions from probability and information theory and by showing that constraint problems can be partially mapped into problems of finding tree-structure joint probability distributions (Chow and Liu, 1968).

### 3.1. *n*-ary relations and probability distributions

Let $P(X)$ be a joint probability distribution of $n$ discrete variables $X_1$, $X_2, \ldots, X_n$. A product approximation of $P(X)$ is defined to be a product of several lower-order distributions (also called marginal distributions) in such a way that the product is a probability extension of these lower-order distributions. A particular class of product approximation considers only second-order components where each variable is conditioned upon at most one variable. The relationships between the variables can be therefore represented by a tree. Given a directed spanning tree of the variables (the direction is from sons to parents) as in



P(X) = P(X1)·P(X2 | X1) · P(X3 | X2) · P(X4 | X2) · P(X5 | X2) · P(X6 | X5)

Figure 11.

Figure 11, the distribution function associated with it is given by the product:

$$P(X) = \prod_{X=(x_1,x_2,\ldots,x_n)} P(x_i \mid x_{p(i)}) \tag{15}$$

$p(i)$ is the parent index of variable $i$. When $P(x_0 \mid x_{p(0)}) = P(x_0)$, 0 denotes the root of the tree. Chow and Liu (1968) had shown that if the measure of distance between two probability distributions $P$ and $P_a$ is given by:

$$I(P, P_a) = \sum_X P(X) \log \frac{P(X)}{P_a(X)} \tag{16}$$

then the closest tree-dependence distribution to $P$ is the one that corresponds to the maximum spanning tree when the weight of each arc is $I(X_i, X_j)$. $I(X_i, X_j)$ is Shanon's mutual information between $X_i$ and $X_j$, defined by:

$$I(X_i, X_j) = \sum_{x_i, x_j} P(x_i, x_j) \log\left(\frac{p(x_i, x_j)}{P(x_i)P(x_j)}\right) \tag{17}$$

$I(P, P_a)$ can be interpreted as the difference of the information contained in $P(X)$ and that contained in $P_a(X)$ about $P(X)$.

Chow's results are remarkable in that a global measure of closeness can be maximized by attending to local measures on individual arcs. We therefore, attempt to adopt Chow's results to our need. Mapping probability distributions to constraints relations, we say that a relation $\rho$ is associated with a distribution function $P_\rho$ if:

$$P_\rho(x_1, x_2, \ldots, x_n) = \begin{cases} 0 & \text{if } (x_1, x_2, \ldots, x_n) \notin \rho \\ 1/|\rho| & \text{otherwise.} \end{cases} \tag{18}$$

Let $\rho_t$ be the relation represented by a constraint tree, $t$, and let $P_\rho$ and $P_{\rho_t}$ be the distributions associated with relations $\rho$ and relation $\rho_t$, having sizes of $l$ and $l_t$, respectively. The 'distance' between the two distributions:

$$I(P_\rho, P_{\rho_t}) = \sum_{X \in \rho} \frac{1}{l} \log \frac{l_t}{l} = \log \frac{l_t}{l} \tag{19}$$

is a monotone function of $l_t/l$ whose inverse was already proposed as a measure of closeness between two relations (where one contains the other). Accordingly, finding the closest tree dependence distribution $P_{\rho_t}$ to $P_\rho$ will result in the closest approximation of a tree relation $\rho_t$ to $\rho$. Equivalently, in order to minimize $l_t/l$ we need to find the maximum spanning tree with respect to the measure $I(X_i, X_j)$. From the given mapping between relations and distributions (equation (18)) we get that:

$$P(x_i, x_j) = \frac{n(x_i, x_j)}{l} \tag{20}$$

$$P(x_i) = \frac{n(x_i)}{l} \tag{21}$$

where $n(x_i, x_j)$ is the number of tuples in $\rho$ having $X_i = x_i$ and $X_j = x_j$, and $n(x_i)$ is the number of tuples in $\rho$ with $X_i = x_i$. Substituting (20) and (21) in (18) we get

$$I(X_i, X_j) = \sum_{x_i, x_j} \frac{n(x_i, x_j)}{l} \log l \cdot \frac{n(x_i, x_j)}{n(x_i)n(x_j)} \tag{22}$$

$$= \log l + \frac{1}{l} \sum_{x_i, x_j} n(x_i, x_j) \log \frac{n(x_i, x_j)}{n(x_i)n(x_j)} \tag{23}$$

consequently the appropriate measure of arc weight is:

$$m(X_i, X_j) = \sum_{x_i, x_j} n(x_i, x_j) \log \frac{n(x_i, x_j)}{n(x_i)n(x_j)}. \tag{24}$$

The question now is how to obtain the quantities $n(x_i)$, $n(x_i, x_j)$ needed for computing $m$. To find them accurately, we need to inspect the list of tuples permitted by the global relation which, of course is unavailable. In the case of probability distributions the marginal probabilities $P(x_i)$, and $P(x_i, x_j)$ are estimated by sampling vectors from the distributions and calculating the appropriate sample frequencies. This cannot be done in our case since finding even one tuple that satisfies the network solves the entire problem. All that we have available is the network of constraints and, therefore, we must approximate the weight $m(X, Y)$ by examining only properties of the arc $(X, Y)$. This leads to approximations:

$$\hat{n}(x_i, x_j) = \begin{cases} 1 & (x_i, x_j) \in r(X_i, X_j) \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

$$\hat{n}(x_i) = N_{X_j}(x_i). \tag{26}$$

Where $N_{X_j}(x_i)$ is the number of pairs in the constraint $r(X_i, X_j)$ with

$X_i = x_i$. Substituting (25) and (26) in (24) we get:

$$m_2(X_i, X_j) = \sum_{(x_i,x_j)\in r(X_i,X_j)} \log \frac{1}{\hat{n}(x_i)\hat{n}(x_j)} \tag{27}$$

$$= -\sum_{(x_i,x_j)} (\log \hat{n}(x_i) + \log \hat{n}(x_j)) \tag{28}$$

$$= -\sum_{x_i} \hat{n}(x_i) \log \hat{n}(x_i) - \sum_{x_j} \hat{n}(x_j) \log \hat{n}(x_j). \tag{29}$$

The behavior of this measure can be illustrated in some special cases:

(a) if the constraint $r(X, Y)$ is the universal constraint (and assuming $k$ values for each variable) then $m_2(u(X, Y)) = -2\sum(k-1)\log k = -2k(k-1)\log k$;

(b) if $r(X, Y)$ is the empty constraint $\Phi(X, Y)$ then we define $m_2(\Phi(X, Y)) = 0$;

(c) if any value of $X_i$ is allowed to go with exactly $r$ values of $X_j$ then $m_2 = -2k \cdot r \log r$. If $r = 1$ we get $m_2 = 0$;

(d) when only one value in one variable is permitted with all the values of the other $m_2 = -k \cdot \log k$.

We see that this measure considers not only the number of the pairs allowed but also their distribution over the $k^2$ slots available. For a uniform constraint—like case (c)—it can be seen that

$$m_2 = -2N \cdot \log r \tag{30}$$

when $N$ is the size of the constraint.

We next give an example showing the behaviour of the accurate measure of weight, $m$, compared with their estimates, $m_2$.

Consider the relation between three binary variables, $X, Y, Z$, given by:

$$\rho = \{(1, 1, 1), (1, 0, 0), (1, 1, 0), (0, 0, 0)\} \tag{31}$$

where the order of the variables is $(X, Y, Z)$. A network representing this relation is given in Figure 12 where the nodes are the variables and the lines correspond to permitted pair of values between pairs of variables. The accurate measures of $n(x_i, x_j)$ and $n(x_i)$ for the pair $(X, Y)$ are given by: $n(0, 0) = 1$, $n(0, 1) = 0$, $n(1, 0) = 1$, $n(1, 1) = 2$, $n(X = 0) =$

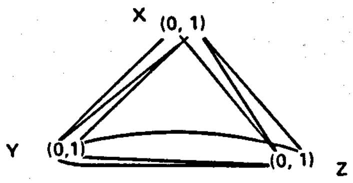


Figure 12.

1, $n(X=1)=3$. Therefore, substituting in (24) we get:

$$m(X, Y) = \log\frac{1}{2} + \log\frac{1}{2\cdot 3} + 2\log\frac{2}{3\cdot 2} = \log\frac{1}{108}.$$

Similarly, for the two other pairs, we obtain:

$$m(X, Z) = \log\frac{4}{729}$$

$$m(Y, Z) = \log\frac{1}{108}.$$

This suggests that the relation may be best approximated by a tree consisting of the arcs $(X, Y)$ and $(Y, Z)$. Indeed, the elimination of the arc $(X, Z)$ will not change the relations at all whereas it is not possible to express $\rho$ by removing either $(Y, Z)$ or $(X, Z)$ only.

By comparison, the network $R$ and (26) give the weight estimates:

$$m_2(X, Y) = -4, \qquad m_2(Y, Z) = -4, \qquad m_2(X, Z) = -4.$$

Which, in this case, fail to distinguish between the various constraints.

In conclusion, we suggest generating tree-approximations for networks using the maximum spanning tree algorithm. Two measures for constraint strength, to be used by the algorithm, are proposed and justified.

## 4. THE UTILITY OF THE ADVICE-GENERATION SCHEME

We compare here the performance of Advised Backtrack (abbreviated ABT) with that of Regular Backtrack (RBT) analytically, via worst-case analysis, and experimentally, on a random constraint problem.

### 4.1. Worst-case analysis

An upper bound is derived for the number of consistency checks performed by the algorithms as a function of the problem's parameters and the number of backtracks performed. A consistency check occurs each time the algorithm checks to verify whether or not a pair of values is consistent with respect to the corresponding constraint.

Let $\#B_A$ and $\#B_R$ be the number of backtracks, and $N$(ABT) and $N$(RBT) the number of consistency checks performed by ABT and RBT, respectively. The problem's parameters are $n$, the number of variables, and $k$, the number of values for each variable. Parameters associated with the constraint graph are $|E|$, the number of arcs, and *deg*, the maximum degree of variables in the graph.

The number of backtracks performed by an algorithm is equal to the number of leaves in the search tree which it explicates. We assume that

$$\text{number of nodes expanded} = c \cdot \#B$$

approximately holds for some constant $c$. (This truly holds for uniform trees where $c$ is the branching factor.) Therefore we use the number of backtracks as a surrogate for the number of nodes expanded. Let $\#C_A$ and $\#C_R$ be the maximum number of consistency checks performed at each node for the ABT and RBT, respectively. We have:

$$N \leq \#B \cdot \#C. \tag{32}$$

Considering RBT first, the number of consistency checks performed at the $i$th node in the order of instantiation is less then $k \cdot deg(i)$. That is, each of this variable's values should be checked against the previous assigned values for variables which are connected to it. We get:

$$N(\text{RBT}) \leq k \cdot deg \cdot \#B_R. \tag{33}$$

The ABT algorithm performs all of its consistency checks within the advice generation. For the $i$th variable, a tree of size $n-i$ is generated. The consistency checks performed on this tree occur in two phases. In the first phases, for each variable in the tree, the values which are consistent with the already assigned values are determined. The consistency checks for a variable $v$ in the tree take $k \cdot w(v)$, where $w(v)$ is the number of variables connected to $v$ which were already instantiated. Therefore, for all variables in the tree

$$k \cdot \sum_{v \in \text{tree}} w(v) \leq k \cdot |E|. \tag{34}$$

The second phase counts the number of solutions. We already showed that the counting takes no more than $(n-1)k^2$ which is bounded by $nk^2$. We get

$$N(\text{ABT}) \leq (k \cdot |E| + nk^2) \cdot \#B_A \tag{35}$$

We want now to determine the ratio between $\#B_A$ and $\#B_R$ for which it will be worthwhile to use Advised Backtrack instead of Regular Backtrack. To do the comparison we will treat the upper bounds as tight estimates while being aware of the possible error. Even though

$$N(\text{ABT}) \leq N(\text{RBT}) \tag{36}$$

is not implied by

$$(k \cdot |E| + nk^2) \cdot \#B_A \leq k \cdot deg \cdot \#B_R, \tag{37}$$

we take (37) as an indicator for the utility of ABT. From (37) we get

$$\frac{\#B_R}{\#B_A} \geq \frac{|E|}{deg} + \frac{nk}{deg}, \tag{38}$$

and, using

$$\frac{|E|}{deg} \leq n, \tag{39}$$

(38) will hold if

$$\frac{\#B_R}{\#B_A} \geq n + \frac{nk}{deg}. \tag{40}$$

Therefore, ABT is expected to result in a reduction in the number of consistency checks only if it reduces the number of backtracks by a factor of $[n + (nk/deg)]$. Thus, the potential of the proposed method is greater in problems where the number of backtracks is exponential in the problem size.

### 4.2. Experimental results

The random CSP instances were generated using a constraint-satisfaction problem generator. The CSP generator accepts four parameters: the number of variables $n$; the number of values for each variable $k$; the probability $p_1$ of having a constraint (an arc) between any pair of variables; and the probability $p_2$ that a constraint allows a given pair of values. As indicated above, it is necessary to keep track of two performance measures; the number of backtrackings ($\#B$) and the number of consistency checks performed. The number of consistency checks gives an indication whether or not the saving in the number of backtracking does not cost too much. What we expect to see is that the more difficult the problems are, the larger are the benefits resulting from using Advised Backtrack.

In our experiments we use $m_1$, the size of the constraint, as the weight for finding the minimal spanning tree in advice. Using the alternative weight, $m_2$, is not expected to improve the results for two reasons. First, the problems generated were quite homogeneous and we have shown that for such problems both weights are the same. Second, the results we get are so good in terms of number of backtrackings that we cannot improve them much by changing the weights.

Two classes of problems were selected. The first with 10 variables and five values, generated with $p_1 = p_2 = 0.5$, and the second with 15 variables and five values, generated with $p_1 = 0.5$ and $p_2 = 0.6$. Ten problems from each class were generated and solved by both ABT and RBT. The order in which the variables were instantiated was determined, for both algorithms, by the structure of the constraint graph. Namely, variables were selected in decreasing order of their degrees (heuristically corresponding to the notion of width developed by Freuder, 1982). The order of value selection is determined by the advice mechanism in ABT. In RBT, the order

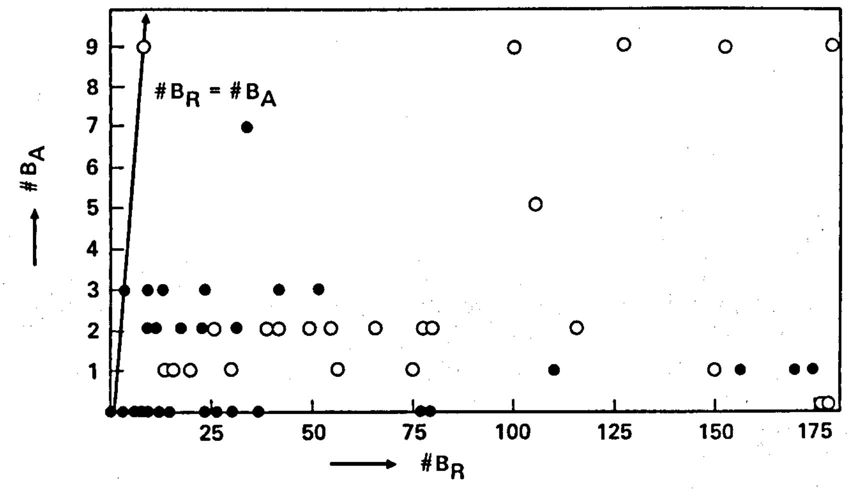


Figure 13.

of value selection was chosen at random. Therefore, while ABT solved each problem instance just once, RBT was used to solve each problem several (five) times to account for the effect of value selection order. When a problem has no solution, the number of backtrackings and consistency checks in RBT is not dependent on the order of value selection, and in these cases the problem was solved only once by RBT.

Figures 13 and 14 display the results of the comparison for both classes

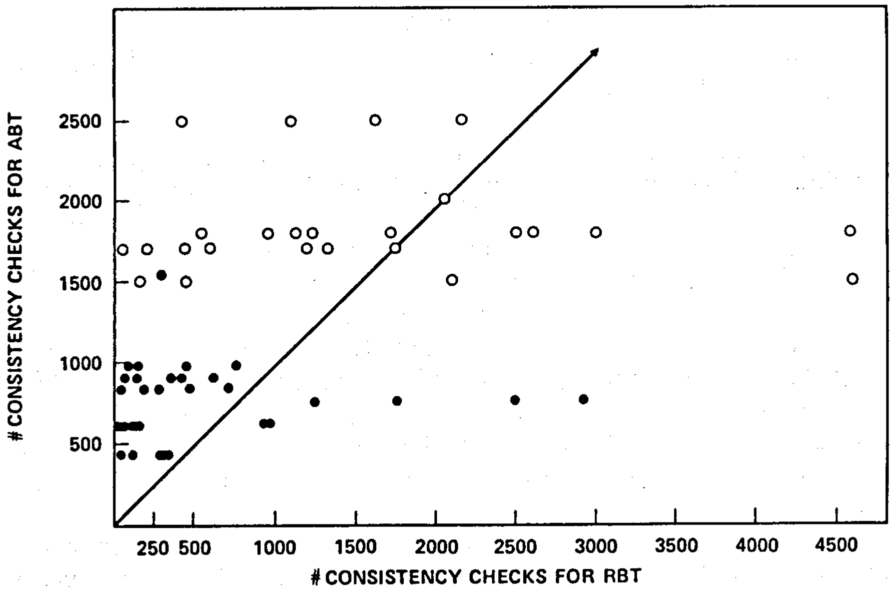


Figure 14.

of problems. In Figure 13, the horizontal axis gives the number of backtrackings that were performed by RBT for each problem instance and the vertical axis gives the number of backtrackings performed by ABT. The points that are indicated by filled circles correspond to problem instances from the first class while empty circles correspond to the second class of problems. We observe an impressive saving in $\#B$ when advice is used for all instances, especially for the second class in which the problems are larger. Figure 14 uses the same method to compare the number of consistency checks. Here, we observe that in many instances the number of consistency checks in ABT is larger than in RBT, indicating that the extra effort in 'advising' backtrack was not worthwhile in those cases.

These results are consistent with the theoretical development of the preceding subsection. If we substitute the parameters of the first class of problems in (40) we get that $\#B_A$ should be smaller than $\#B_R$ by at least a factor of 20 (25 for the second class of problems) for us to expect an improvement in the performance. Many of the problems however, were not hard enough (in terms of the number of backtrackings required by RBT) to achieve these levels.

In Figure 15 we give the comparison between the two algorithms only for problems that turned out to be difficult. We display the number of consistency checks of problems from both classes in the cases where the number of backtrackings in RBT were at least 70. We see that the majority of these problems were solved more efficiently with ABT than with RBT.

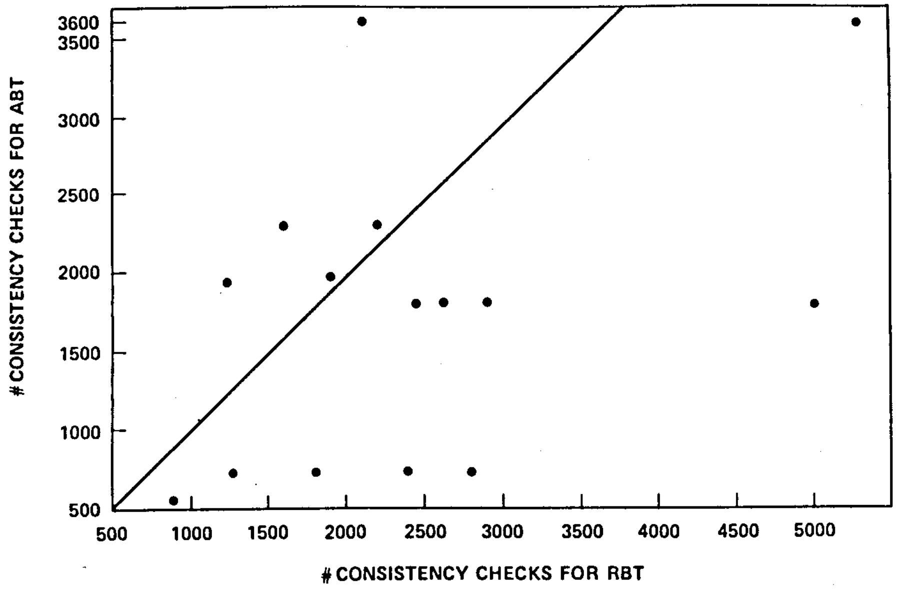


Figure 15.

Experiments were also performed on the $n$-queen problem for $n$ between 6 and 15 and on the three-colourability problem on a set of random graphs. In all cases the number of backtrackings of ABT was smaller than RBT, but the problems were not difficult enough to obtain a net reduction in the number of consistency checks.

Experiments related to the ones reported here were performed by Haralick and Elliot (1980). The forward-checking lookahead mechanism, reported to exhibit the best performance considering the number of consistency checks, can be viewed as an automatically generated advice in the sense discussed here. However, since Haralick and Elliot are interested in finding all solutions to CSP, and we deal with finding just one solution, the results cannot be directly related.

As a conclusion, advice should be invoked on problems which are hard for RBT. Therefore one needs a way of recognizing that a problem instance is difficult. For example, Knuth (1975) has suggested a simple sampling technique that requires very little computation to estimate the size of the search tree. These estimates can be used in conjunction with parametrized advice that adapts itself according to the expected size of the tree. Namely, smaller problems may benefit from a weaker advice (or no advice at all) which may not be as good but is more efficient.

## REFERENCES

Arnborg, S. (1985) Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT* **25**, 2–23.

Bertele, U. and Brioschi, F. (1972). *Nonserial dynamic programming.* Academic Press, New York.

Carbonell, J. G. (1983) Learning by analogy: formulation and generating plan from past experience. In *Machine learning* (eds Carbonell, J., Michalski, R. S., and Mitchell, T.) Tioga, Palo Alto.

Chow, C. K. and Liu, C. N. (1968) Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory* **14**(3), 462–467.

Even, S. (1979) *Graph algorithms.* Computer Science Press, Maryland.

Freuder, E. C. (1982) A sufficient condition of backtrack-free search. *J. Association for Computer Machinery* **29**, 24–32.

Gaschnig, J. (1979) A problem similarity approach to devising heuristics: first results. *Proc. 6th International Joint Conf. on Artificial Intelligence,* Tokyo, pp. 301–307.

Guida, G. and Somalvico, M. (1979) A method for computing heuristics in problem solving. *Information Sciences* **19**, 251–259.

Haralick, R. M. and Elliot, G. L. (1980) Increasing tree search efficiency for constraint satisfaction problems. *AI Journal* **14**, 263–313.

Knuth, D. E. (1975) Estimating the efficiency of backtrack programs. *Mathematics of Computation* **29**, 121–136.

Mackworth, A. K. (1977) Consistency in networks of relations. *Artificial Intelligence* **8**, 99–118.

Mackworth, A. K. and Freuder, E. C. (1985) The complexity of some polynomial consistency algorithms for constraint satisfaction problems. *Artificial Intelligence* **25**(1), 65–74.

Mohr, R. and Henderson, T. C. (1986) Arc and path consistency revisited. *Artificial Intelligence* **28(2),** 225–233.

Montanari, U. (1974) Networks of constraints: fundamental properties and applications to picture processing. *Information Science* **7,** 95–132.

Nudel, B. (1983) Consistent labelling problems and their algorithms: expected complexities and theory based heuristics. *Artificial Intelligence* **21,** 135–178.

Pearl, J. (1983) On the discovery and generation of certain heuristics. *AI Magazine,* 22–23.

Purdom, P. W. and Brown, C. A. (1985) *The analysis of algorithms.* CBS College Publishing, Holt, Rinehart & Winston.

Sacerdonti, E. D. (1974) Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* **5,** 115–135.

Simon, H. A. and Kadane, J. B. (1975) Optimal problem solving search: all or none solutions. *Artificial Intelligence* **6,** 235–247.

7

# The Relation Between Programming and Specification Languages with Particular Reference to Anna

A. D. McGettrick and J. G. Stell
Department of Computer Science,
Strathclyde University, Glasgow, UK

**Abstract**
The development by the US Department of Defense of the programming language Ada has provided a language which is certain to become widely used. One of the design criteria for Ada was that it should encourage the production of reliable software. Many remedies have been suggested for the problem of software reliability; however, it seems vital to have a specification of the function a program is to perform before it is actually constructed. If a specification is written in a formal language, a language for expressing more abstract concepts than the programming language, it can be both semantically precise and amenable to manipulation by a computer.

The Anna language is an extension of Ada that was designed to permit the annotation and specification of Ada programs. The design is based on the Ada notation and it is intended that the language should support all aspects of the program development process.

Anna has been developed over a number of years. Preliminary notification of the existence of the language was given at an Ada Symposium in December 1980 [see ref. 2] superseded in March 1984 by the latest document, the Preliminary Reference Manual [3].

In this paper we shall look at a number of issues associated with the design of a language such as Anna.

## 1. BACKGROUND—THE STANFORD PASCAL VERIFIER

The activity associated with the Stanford Pascal Verifier (SPV) has had a profound influence on such topics as program verification, program specification and even programming methodologies. As originally conceived, this was an academic exercise designed to unearth and then study problems associated with the implementation of tools to support the verification process. It was never intended that the SPV system would be of production quality or a commercial product.

We shall highlight the lessons learned from the SPV exercise, indicate the developments that led to the language Anna, and then make a brief assessment of Anna.

## 1.1. The Stanford Pascal Verifier

A typical input to the SPV system would be

```
PASCAL
VAR N, R;
PROCEDURE MULTIPLY (VAR X: INTEGER; Y; INTEGER);
GLOBAL (N);
EXIT X = Y * N;
VAR Z: INTEGER;
BEGIN
  X< -0; Z< -0;
  INVARIANT X = Y * Z;
  WHILE Z = N DO BEGIN X < -X + Y;
                       Z< -Z + 1
                 END
END;
EXIT R = N * N;
BEGIN
  MULTIPLY (R, N)
END.
```

(See ref. [1], p 14).

Here EXIT statements indicate the results, outcome or postcondition resulting from the execution of the block they precede. GLOBAL($N$) indicates that $N$ is a constant and will be accessed within the procedure MULTIPLY. INVARIANT identifies the loop invariant of the loop that it precedes.

This piece of text can be submitted to the SPV system and successfully processed by it, thereby granting a kind of approval.

The reader will immediately make a number of observations, some of which indicate drawbacks to the system.

(1) The program may never terminate so that the SPV addresses questions of partial correctness and not total correctness (indeed $N$ has not even been initialized in this program).

(2) The program is written not in Standard Pascal but in a variant of Pascal, not processable by any popular compiler; consequently wide-ranging modifications need to be made.

(3) The presence of GLOBAL, EXIT, INVARIANT, etc indicate the need for a language which is quite separate from PASCAL and which constitutes an annotation or documentation language.

These observations are superficial but they indicate the presence of some serious points. It is generally no simple matter or cosmetic exercise to translate from a working program to input that might be submitted for very careful scrutiny by the SPV system. Conversely text that has been successfully processed by the SPV may not run satisfactorily.

## 1.2. Implementation and use

The SPV system can be thought of as being implemented in a number of phases.

A *verification condition generator* accepts input of the kind described above, i.e. a program and its accompanying documentation. From this, a set of verification conditions are generated and these are then supplied as input to a *theorem prover*. The theorem prover attempts to take the verification conditions and establish their truth, thereby indicating that the program conforms to its specification.

The theorem prover can be viewed as two separate entities: a simplifier carries out algebra associated with simple data types; the rule handler deals with more complex data structures and can accept axioms of the form

GCD1: REPLACE GCD $(X, 0)$ BY $X$:
GCD2: REPLACE GCD $(X, X)$ BY $X$:
GCD3: REPLACE GCD $(X, Y)$ BY GCD $(Y, X)$
GCD4: REPLACE GDC $(X, Y)$ WHERE $Y > 0$ BY GCD (MOD$(X, Y)$, $Y$);

from a file.

In the event of the theorem prover successfully processing its input, then there is little left to concern the user other than the limitations already identified. However, if the theorem prover fails to complete the task the user of the SPV system has then to resolve the conflict. This can occur because:

(a) of an incompatibility between the program and its accompanying documentation;

(b) the system has not been supplied with enough information about the items being processed.

In both cases, changes must be made either to the program, to its documentation or to accompanying information (e.g. as provided for GCD).

The process outlined above is then repeated. Thus a complete set of verification conditions are generated and these are again reprocessed by the prover.

Some criticisms can be made of the approach described above:

(a) the repeated regeneration of all verification conditions will usually be unnecessary;

(b) when errors occur a user is confronted with a verification condition which usually bears little resemblance to the original program;

(c) there is no check on the accuracy of axioms such as those provided for GCD.

## 2. THE ANNA LANGUAGE

### 2.1. Introduction

Anna [3] is a language for annotating Ada programs (hence its name). These annotations take the form of formal comments. Syntatically these are distinguished from informal Ada comments by being introduced by the characters – –: or – –| rather than only – –. There are two classes of formal comments: annotations and virtual Ada. The annotations are generally logic formulas concerning program variables and include the usual assertions used in the Floyd–Hoare style of program verification. Virtual Ada, on the other hand, is syntactically Ada text which may be used to define functions, for example, which themselves may be used in the annotations. The virtual Ada may itself be annotated in exactly the same way as the underlying actual Ada.

Before looking at some of the details of the form annotations may take, we consider how the semantics of Anna are defined and how this relates to machine aids for checking consistency of Anna. A formal definition of Anna may be given in terms of axioms and proof rules. However the preliminary Anna manual gives an alternative definition in terms of transformation rules and the Anna kernel. The Anna kernel consists of a subset of Anna taking a particularly simple form; principally it contains assertions. Assertions are defined to be those annotations which constrain only one program state, that is those logical formulas whose truth or falsity depend on the values of program variables at only one point of a computation, e.g. $x = 1$ in

```
x := 1;
– –| x = 1
y := 1;
```

Many annotations which constrain several states can be replaced by equivalent assertions. For example an annotation constraining the value of a variable to lie within certain limits may be replaced by suitable assertions at every place where the variable values might be altered.

When Anna programs consist only of annotations which may be reduced to assertions there is the possibility of checking consistency at run time without the need for formal proof. Since each assertion constrains only one state it is possible to replace each assertion by actual Ada code which tests whether the variables have permissible values at that point in the program. If an inconsistency is detected the exception ANNA_ERROR is raised. While such run time checking is not as powerful as a formal proof or as useful, since it gives no information about the program's reliability in general, it is relatively straightforward to implement as an extension to an Ada compiler.

### 2.2. Annotations on objects and types

We now examine some of the annotations defined in Anna, starting with object annotations. Object annotations constrain the values of program verification. Virtual Ada, on the other hand, is syntactically Ada text which may be used to define functions, for example, which themselves may be used in the annotations. The virtual Ada may itself be annotated in exactly the same way as the underlying actual Ada.

```
M, N: INTEGER := 0;
- - | N ≤ M;
```

constrains $N$ to be less than or equal to $M$. Next,

```
X: INTEGER := 0;
- - | 0 < X < 100;
```

is an example of an object constraint which could be expressed in Ada by using a subtype declaration. This could also be done by using an Anna annotation on a type

```
subtype first_hundred is INTEGER;
- - | where Z: first_hundred ⇒ 0 ≤ Z ≤ 100;
X: first_hundred;
```

Not all Anna constraints on types are expressible as Ada constraints, for example:

```
subtype EVEN is INTEGER;
- - | where Z: EVEN ⇒ Z mod 2 = 0;
```

### 2.3. Statement annotations

These are the most familiar types of annotation to those experienced in verification.

Simple statement annotations constrain only one statement and appear immediately after it as in:

```
X := X + 1;
- - | X = in X + 1;
```

The meaning of such an annotation is determined by elaborating it before the statement to which it refers. For the checking of consistency the value of *X* can be compared with the value at the end of the execution of the statement. In a similar way more complex statement annotations can be replaced by run-time checking code.

Compound statement annotations constrain all states in the execution of a compound statement. They can be expressed as follows:

```
  --| with x>y;
  begin
    x:=x+1;  --(i)
    y:=y+1;  --(ii)
  end;       --(iii)
```

which asserts $x>y$ at each of the points marked (i), (ii), (iii).

#### 2.4. Annotations of subprograms

The behaviour of a function subprogram can be specified by a result annotation such as:

```
  function SQUARE (N: INTEGER)  return  INTEGER
  --| where return  N*N;
```

In general procedures can be dealt with using annotations on formal parameters. This allows us to state both input and output specifications relating the actual parameters when any procedure is called. Thus in particular partially developed code using such procedures can be verified subject to correct implementation of the procedures. The ability to discuss these issues is central to a rigorous stepwise development of Ada programs.

#### 2.5. Package annotations

In Ada, packages are the means by which related program entities are grouped together, for instance a collection of operators related to some type. Ada packages have two parts, a specification and a body. The specification provides only syntactic information about the objects introduced in the body. Anna provides the means semantically to characterize these objects, that is to specify their behaviour to a user of the package without such a user needing to examine the package body to determine these properties. The specification of packages is by means of package axioms in the style of much work on algebraic specification.

## 3. SOME OBSERVATIONS

It should be clear that many of the earlier criticisms of the SPV system no longer apply to the Ada/Anna combination. Formal comments help in this respect. Yet much will depend on implementation issues: in this connection interactive systems can be used to overcome much of the unnecessary repetition inherent in the SPV approach. But let us look at other issues.

With the combination of Ada and Anna it is possible to seemingly describe the same concept in a variety of ways.
Consider

```
    M:INTEGER  range  1...99;
and
    M:INTEGER; − −| 0<M<100;
```

Of these the first is pure Ada, whereas the second involves the presence of an annotation. This indicates a degree of overlap between the Ada and Anna and raises questions about where the boundary between a programming language and a specification language should lie. In this particular case, Anna provides a more powerful and useful mechanism and this suggests that the Ada mechanism should not be present.

In the same way, the following two specifications appear to be equivalent.

```
generic
  type element is private;
  function compose (a, b: element) return element;
package semi-grouped is
  − −| axiom for all a, b, c: element =
  − −| compose (a, compose (b, c)) = compose (compose (a, b), c)
end semi-group;

generic
  type element is private;
  function compose (a, b: element) return element;
  − −| axiom for all a, b, c: element =
  − −| compose (a, compose (b, c)) = compose (compose (a, b), c)
package semi-group is
end semi-group;
```

The choice between these alternatives requires considerable insight into the nature of the items and packages. The insight is mathematical in nature.

The example of a semi-group provides a convenient source of further attention. Suppose we try to utilize this in the following way:

```
package group is
  use semi-group;
  identity: element;
  function inverse (a: element) return element;
  − −| axiom for all a: element ⇒
  − −| compose (a, identity) = a,
  − −| ...
end group:
```

Several points follow from this:

(a) the concept of a semi-group can be extended or enriched to produce the concept of a group; more constraints are imposed;

(b) because of the nature of Ada, and the fact that package bodies must accompany package specifications, the structure of an implementation must follow rather closely the structure of the specification;

(c) sharing a specification, e.g. of semi-group, but permitting different implementation is not possible.

Of these (a) and (b) should be noted, and may be seen as disadvantages of the Anna approach.

In summary, Anna can be seen in a variety of lights—as a specification language or as an annotation language. But further it contributes with Ada to a wide spectrum language which supports specification at one end, programs at the other and permits the possibility of transformations between specifications, between specifications and programs, and even between programs.

At the present time the definition of Anna is incomplete and further developments are expected. These will address complex issues such as concurrency.

## REFERENCES

1. Luckham, D. C., German, S. M., Henke, F. W. V., Karp, R. A., and Milne, P. W. (1979). The Stanford Pascal Verifier User Manual. Stanford Verification Group. Report no. 11. Stanford University, Stanford, Calif.
2. Krieg-Brueckner, B. and Luckham, D. C. (1980). Anna: Towards a language for annotating Ada programs. *ACM SIGPLAN Symposium on the Ada programming language.* SIGPLAN Notes 15, 11, 128–135.
3. Luckham, D. C., Henke, F. W., Krieg-Brueckner, B., and Oue, O. (1984). Anna, A language for annotating Ada programs. Technical Report no. 84–248. Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, Calif.

# LOGIC PROGRAMMING TOOLS AND APPLICATIONS

# 8
# YAPES: Yet Another PROLOG Expert System

T. B. Niblett
The Turing Institute and University of Strathclyde,
Glasgow, UK

## 1. INTRODUCTION

YAPES is an expert system shell developed at the Turing Institute. It provides inference and explanation facilities, and incorporates a novel form of plausible inference.

YAPES is a specialized *interpreter* for logic programs. Figure 1 illustrates its top level structure. A PROLOG interpreter (or compiler) executes such programs consisting of sets of *Horn clauses,* a form of first-order logic. The YAPES system also executes such programs, as well as programs in an extended version of Horn clause logic which uses *certainties* as truth values, rather than just true and false.

The construction of the YAPES interpreter allows it to provide explanations of its reasoning, as well as asking questions from users and other knowledge sources. The explanations provided include *why explanations* in the sense of 'why are you asking this question', *how explanations* in the sense of 'how did you reach this conclusion' and *why not explanations* in the sense of 'why did this goal fail'.

The YAPES interpreter is flexible and can call the PROLOG interpreter to execute goals directly. Similarly the PROLOG interpreter can call the YAPES interpreter. This allows a mixture of *object* and *meta-level* inference, and provides a smooth integration with the PROLOG environment.

In Section 2 the rationale behind YAPES is described in more detail and the advantage of using logic as a representation language discussed. Section 3 describes the facilities provided by YAPES, and contains some discussion of the theoretical issues involved. Section 4 presents an annotated transcript of a YAPES session using a simple knowledge base for travel expense claims. Finally in Section 5 we discuss the advantages of the approach taken.

Many of the ideas incorporated in the program are not new and derive much from the work of Kowalski (1979), Shapiro (1983b), Sergot (1983) and others. In particular many of the utilities incorporated in the

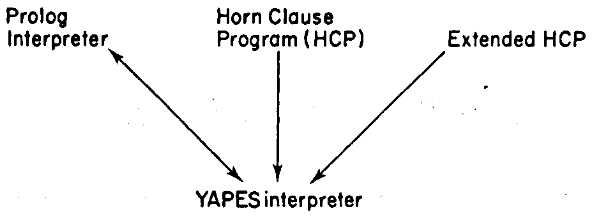


Figure 1. YAPES as an interpreter for logic programs.

program were written by Richard O'Keefe and Lawrence Byrd and I am greatly indebted to them.

## 2. THE RATIONALE BEHIND YAPES

Our intention has been to provide an expert system 'shell' containing all the normal trace and explanation facilities, and with the provision of a sophisticated plausible reasoning facility.

The best way of looking at YAPES is as a new interpreter for rather special PROLOG programs which provides the kind of user interaction needed in expert systems work.

The program is implemented in PROLOG and a major concern has been to keep the form of the knowledge-base as close to the underlying language (Horn clause logic) as possible. In fact the only extension to pure Horn clause logic is the use of a plausible reasoning mechanism, which preserves the clear semantics of the logic and is *upwards compatible* in the sense of being a conservative extension. There are several compelling reasons for taking this approach:

1. The semantics of logic programs are well understood. This permits the development of elegant and general algorithms for debugging and modifying programs which can be directly applied to expert system knoweldge-bases (e.g. Shapiro, 1983b).

2. Most commercial expert systems shells available at present lack flexibility. It is often difficult to modify the search strategy used by the shell, or to perform computations not explicitly catered for by the designers. Logic programming is far more general and by keeping our shell as close as possible to logic programs this generality can be incorporated while still allowing the system debugging and tracing facilities to be used.

3. By using the same language for the knowledge-base and the underlying program we can take advantage of the reflection principle espoused by Weyhrauch (1980) and merge the *object* and *meta-level* languages where appropriate.

### 2.1. Logic programs as a basis for expert systems

The basis of our approach is that a logic program, and the results of its execution in the form of a proof tree (see below) can be used as a knowledge-base and can provide humanly understandable explanations of how the knowledge is used by the system. Furthermore the simplicity and generality of the 'data-structures' involved allow an inexperienced user to learn the use of the system fairly quickly.

We are *not* arguing that the PROLOG language is easy to learn in its full generality. From our experience this is not the case. For the majority of *expert system* applications however, where the manipulation and creation of complex data structures is not required, the language is very straightforward.

### 2.2. The form of a logic program

We shall discuss fairly briefly the form of logic programs; this material is useful for a technical appreciation of the plausible reasoning mechanism discussed later, but not essential for the reader who wishes merely to discover what YAPES *does*.

A logic program consists of a (non-empty) set of *Horn clauses*, together with a *goal statement*. A Horn clause has the form $A \leftarrow B_1, .., B_n$ where $A$ and $B_i$ are *literals*. A literal is of the form $p(T_1, \ldots, T_m)$ $(m \geq 0)$where $p$ is a *predicate name* and the $T_i$ are *terms*. A term is a *variable*, an *atom*, a *number* or an expression of the form $f(T_1, \ldots, T_m)$ $(m > 0)$ where $f$ is a *function symbol* and the $T_i$ are terms. We shall follow the PROLOG convention throughout that variables are indicated by an initial capital letter, and predicate names, function symbols and atoms by an initial lower-case letter. The goal statement of a program is written $\leftarrow G$ where $G$ is a literal.*

A Horn clause $A \leftarrow B_1, \ldots, B_n$ is interpreted in first-order logic as the assertion

$$\forall_{x_1}, \ldots, x_n\{B_1 \;\&\; \cdots \;\&\; B_n \rightarrow A\}$$

where $\rightarrow$ is material implication and $x_1, \ldots, x_n$ are variables occurring in the literals of the clause.

### 2.3. Unification

The central operation in logic programming is *unification*. Unification occurs between two literals, and involves finding a (possibly empty) substitution for the variables in the literals which makes them identical. For example the literal $f(g(X), h(h(Y)))$ unifies with the literal $f(U, h(V))$ with the substitution set $\{g(X)/U, h(Y)/V\}$. It has the useful

* A conjunction of literals $\leftarrow G_1, \ldots, G_n$ is often used in PROLOG. This is equivalent to having a single goal $\leftarrow G$ and adding the pseudo-goal $G \leftarrow G_1, \ldots, G_n$.

property in practice of allowing us to abstract the *control* of a logic program from the *flow of data* as we shall see later.

### 2.4. Execution of a program

A logic program executes successfully if the initial goal statement $\leftarrow G$ can be reduced to the empty goal ($\leftarrow$). This reduction is accomplished in stages as follows; at each stage we have a goal statement $\leftarrow G_1, \ldots, G_n$, one such goal statement $G_i$ is chosen* and *unified* with the head of one of the clauses in the program. If for the chosen goal $G_i$ there is more than one clause that can be unified, we have a *choice point.* If no $G_i$ can be unified with any clause then the execution must backtrack to a previous choice point, where an alternative clause can be found to match. This unification produces a substitution for the variables in the goal statement and in the body of the clause concerned. The goals in the body of the clause are then substituted in the goal statement in place of the goal $G_i$. Execution terminates successfully when there are no goals left.

Figure 2a demonstrates the execution of a simple program without variables. Figure 2b illustrates how this execution can be displayed as a

Program

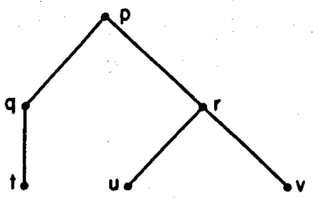
```
p:-q,r.
r:-u,v
q:-s.
q:-t.
u.v.t.
```

Execution trace

```
call p
          call q
                    call s
                    fail s
          retry q
                    call t
                    exit t
          exit q
          call r
                    call u
                    exit u
                    call v
                    exit v
          exit r
exit p
```

(a)

Proof tree



(b)

Figure 2

tree. We shall call such trees *proof trees.* We can see that information about the order in which unifications took place is omitted from such trees. Also any information about unifications which were tried and later had to be undone and another choice of clause tried has been omitted.

* In PROLOG the goal chosen is always the leftmost goal $G_1$.

This is one very useful property of logic programs. The output of a successful execution has only the relevant information about a proof: all information about how the proof tree was constructed is omitted. This ability to extract only relevant information is of great utility when it comes to providing explanations in an expert system context.

In the next section we shall see how the particular properties of the PROLOG interpreter also allow us to provide information to the user during execution.

## 2.5. Reasoning about program behaviour

As we have seen, the result of a program's execution is a proof tree. The PROLOG language allows us to create and manipulate such trees by virtue of its *meta-logical* features.

These features allow us to regard clauses and literals as terms, and to determine the current status of PROLOG objects to see whether variables are instantiated or not. In addition we can modify a program during its execution by creating new clauses (and terms, etc.).

This ability of PROLOG to create an internal representation of itself is critical in creating an expert system. Of course it is possible to do this in any language. One can for example create a PASCAL interpreter in PASCAL and simulate the execution of PASCAL programs with it. There are three important differences however.

1. The representation in PROLOG is much shorter, and therefore much easier to work with. It is possible to write a useful PROLOG interpreter in PROLOG in less than one page of code. To do this in PASCAL would be a major achievement.

2. The computation in PROLOG is easily representable as we have seen. It would not be possible to do the same in a procedural language which relies largely on side-effects.

3. The use of unification allows us to write interpreters for logic programs in PROLOG that reveal the important information, about control flow and how data-structures are to be created, without having to describe in detail the way data will be manipulated.

The program in Figure 3 illustrates an interpreter in PROLOG for PROLOG which maintains a stack of goals indicating the branch of the proof tree that the interpreter is currently investigating. This allows a user to question the system as to why it is pursuing a particular goal. This program is a complete interpreter for pure PROLOG (without the cut). Built-in system predicates are directly executed via the *call* predicate. The PROLOG system knows which goals are system goals via the *system* predicate. The meta-level predicate clause returns a clause in the program. Its two arguments are the head and body of the clause respectively. The clauses found are pushed onto the trace list (acting as a stack) as terms.

```
solve(X) :-
    solve(X, []).
solve((A, B), Trace) :-
    !,
    solve(A, Trace),
    solve(B, Trace).

solve(not X, Trace) :-
    not solve(X, [not X | Trace]).

solve(Goal, Trace) :-
    system(Goal),
    !,
    call(Goal).

solve(Goal, Trace) :-
    clause(Goal, Body),
    solve(Body, [(Goal :- Body) | Trace]).
```

Figure 3. Interpreter with trace for PROLOG.

It is possible to write a shorter interpreter for PROLOG, e.g. the single clause *solve(X)←call(X)*. The program of Figure 3 captures just sufficient of the control flow to be useful.

**2.6. The problem of flexibility**

It is frequently observed that expert system shells are inflexible. This problem arises from the tension between requiring a program to represent knowledge and to explain itself and also change the representations used by the program and its control structures fairly frequently. It is difficult to design a programming language which can meet both these requirements.

Our view is that this problem is largely one of the distance between the underlying implementation language of the shell, and the representation language interpreted by the shell. In YAPES a radical solution is provided by using a uniform representation language. The shell is written in the language used to represent knowledge, namely Horn clause logic. Our problem then becomes one of modifying the core interpreter (as shown in Figure 1) and extending, if necessary, the meta-logical control language used by the interpreter.

Our experience with YAPES shows that this is not an intractable problem due to the power and flexibility of unification. It does not take long to modify the interpreter simply because the code is compact. The main interpreter for YAPES contains only eight clauses.

It could be argued that the reason for this is that the control structure of YAPES is very similar to that of PROLOG. This does not seem to be the case. Shapiro (1983a) has published a co-routining interpreter for *Concurrent* PROLOG, written in PROLOG which consists of only 50 lines of code (32 clauses). Equally as important, the very different control structure used by an interpreter such as this does not affect the explanation capabilities in terms of proof trees discussed above, or the extensions to the language used for plausible inference. All this is carried over unchanged: the system is modular.

### 2.7. Implementation

A final practical point should be made at this point about the practicalities of the use of PROLOG. There is little point producing a system, however elegant and simple. if it is impractical to run it in the computing environment of typical users.

Historically PROLOG has suffered from the problem of only being available, in efficient implementations, on a small number of machines—notably the DEC 10/20 range for Edinburgh PROLOG. YAPES has been run successfully with relatively small knowledge-bases (<100 rules) under C-PROLOG on a VAX-750. Recently new versions of PROLOG (e.g. Quintus PROLOG) have become available which have over an order of magnitude gain in speed and space over C-PROLOG. These PROLOGS are available on a wide range of (relatively) cheap hardware and are capable of supporting very large knowledge-bases for YAPES, as well as providing the efficient interface to the underlying operating system which would be necessary in any commercial use of such expert systems.

## 3. THE FACILITIES OF YAPES

The facilities available in YAPES can be split conveniently into three parts.

1. Extensions to the language of Horn clause logic.
2. Interaction with the user.
3. Checking the knowledge-base.

### 3.1. Extensions to the language of Horn clause logic

It is generally recognized that the ability to reason about uncertainty is very useful in an expert system. Unfortunately it is not clear how this should be done.

The first expert system to use plausible reasoning was MYCIN (Shortliffe, 1976) which used a combination of probability theory and the theory of confirmation.

Many attempts have been made in the past decade to improve on the MYCIN formalism, which was recognized to have many defects. Most of these attempts have centred on the problem of assigning probabilities to hypotheses/events. The main issue is one of complexity. Given $N$ events a fully specified probability distribution requires in the order of $2^N$ constraints.

Attempts to apply constraints by making *a priori* assumptions about distributions (e.g. Prospector, Duda *et al.*, 1979) have found that such assumptions are often incorrect and/or unduly restrictive. Attempts such as in Cheeseman (1983) to estimate distributions using entropy measures run into problems of computational complexity.

A further, perhaps more fundamental, problem is that in many circumstances probabilities seen as numbers (or ranges of numbers) are not appropriate in a knowledge-based system—they are too opaque.

In YAPES an attempt is made to solve these problems by shifting our theoretical base and providing a richer domain for certainties than the real number line. We also provide a well-founded semantics that avoids problems of computational complexity.

We proceed by extending the domain of values that literals can take from {*true, false*} to *lattices* of values.

The detailed theory describing how lattices of truth values can be integrated into our logic is described in Appendix A. Here we shall concentrate on the specific system currently used in YAPES. We should emphasize that the system described below is only one of many that could be chosen, and that YAPES is *modular.* A wide variety of inference systems could be quickly implemented.

We have departed from the use of numerical certainties and use what are termed *justifications.* An example of a justification is shown in Figure 4. The literal *entitledToAllowance* (*tim, 52*) is *justified* by the structure below it, which is a justification. This justification is a forest of justifications. The first three are units, the fourth is itself a tree.

Every literal now contains a justification, rather than the (implicit) value *true* as before. The empty justification [] corresponds to *true.*

The use of justifications allows us to control the reasoning of the system more fully. We can specify that two justifications are incompatible for example, and give a low overall certainty to their combination. The

**Tim is entitled to an allowance of fl.52**
is justified if
    **The trip was in Holland**
    and
    **tim didnt go on a boat**
    and
  **The plane cost is more than the car cost in Holland**
    is justified if
      **The trip was in Holland**
  A justification expressed in English. The literal at the top level is *entitled to Allowance* (*tim,* 52)

Figure 4. A justification.

main use at present in YAPES is to reduce the number of questions asked of the user, by making assumptions about the truth of certain literals. If the user dislikes any of the assumptions made these can be overridden by use of the threshold mechanism described below.

We shall now discuss in more detail the calculus of justification currently used by YAPES.

### 3.1.1. *A calculus of justification*

Figure 5 illustrates a (simplified) version of the top-level YAPES interpreter, which calculates certainties. The omissions are intended to make the structure of the mechanism clear while omitting details such as the user interface. A full listing of this interpreter is provided in Appendix B.

The interpreter uses the goal *solve/3* with three arguments. The first argument is the goal to be solved, the second is the *threshold* setting the minimum value for a justification. *Solve* will fail if the goal cannot be solved with at least this certainty. The third argument is the *certainty* of the goal.

A clause is now a pair $\langle just, G \leftarrow B \rangle$ where $G \leftarrow B$ is a Horn clause and *just* is the name of its associated certainty function. The certainty of $G$ is determined from the certainty of $B$ using *just*. The empty

```
solve((A, B), Thresh, Value)
   :- !,
      solve(A, Thresh, Aval),
      solve(B, Thresh, Bval),
      combined_and(Aval, Bval, Value),
      less(Thresh, Value).

solve(not Goal, Thresh, [])
   :- !,
     not solve(Goal, Thresh, _).

solve(Goal, Thresh, [])
   :- system(Goal),
      !,
      call(Goal).

solve(Goal, Thresh, Value)
   :- dataclause(Just, Goal, Body),
      solve(Body, Thresh, BodyVal),
      Combine_if(BodyVal, Just, Value),
      less(Thresh, Value).
```

Figure 5. An interpreter using certainties.

justification (corresponding to truth) is []. Two procedures are used to determine the certainty of $G$. One to determine the certainty of $B$ as a conjunction of goals $B_i$ (combine_and), and one to determine the value of $G$ given a value for $B$ and *just* (combine_if).

The implementation of thresholding imposes a restriction on derivations. Given a threshold $T$ for the top-level goal $G$, every subgoal of $G$ in the proof tree must have a certainty of at least $T$. This reflects the intuitive idea that the certainty of a conclusion cannot be greater than the certainty of any of its premises. In practice this greatly reduces the amount of search necessary to derive the certainty of a goal.

The *default* for certainty functions in the present implementation is to combine the justifications for each of the $B_i$ into a tree with *just* as the root and the certainties of the $B_i$ as subtrees. If the user wishes to override these defaults he/she must write explicit code for the combining functions in any particular case. Similarly a default ordering is imposed on justifications. Given two justifications $A$ and $B$ we say that $A \leq B$ if the set of nodes of $B$ is a subset of the nodes of $A$. The minimal element *any* is also defined such that $any \leq A$ for all $A$.

The user must ensure that the relevant *monotonicity* properties are satisfied when creating defaults. The requirement that the default be monotonic says that if any goals in the body of a clause become more certain then the certainty of the conclusion should not become less likely. This is, on the whole, an intuitively natural condition, the technical reasons for it are described in Appendix A.

### *Expressive power*

There are some circumstances where the user's knowledge about a domain suggests that the above monotonicity requirements are not met. An example, using propositional variables, shows such a situation. Let us assume that we have a goal $h$ and two contributing factors $e_1$ and $e_2$. We may know that either $e_1$ or $e_2$ alone support $h$, but that their joint presence does not. If we have a Horn clause rule: $h \leftarrow e_1, e_2$ in our knowledge-base the monotonicity criterion fails to hold, as either $e_1$ or $e_2$ becomes more likely as the likelihood of $h$ decreases.

This type of constraint cannot be directly expressed within our certainty calculus. To see why we can consider the problem of expressing the exclusive or of two goals $e_1$ and $e_2$ in PROLOG. We wish to say that $h$ is true if $e_1$ is true and $e_2$ is false, or if $e_1$ is false and $e_2$ is true. This corresponds to the problem described above, except that we are considering truth-values rather than likelihoods. Horn clause logic, and therefore a PROLOG interpreter cannot express negation directly and must use the idea of 'negation as failure'; a goal is false if it cannot be proved true. The PROLOG solution to the problem of exclusive or would be (using the unary predicate not to express negation as failure):

$$h \leftarrow e_1,\ not(e_2), \quad h \leftarrow e_2,\ not(e_1).$$

A solution of this form can be used within our certainty calculus. To see how this works we shall first describe its workings from a slightly different viewpoint.

The standard PROLOG interpreter can be understood as producing a proof tree that provides a proof of the top-level query to PROLOG. The value returned by this proof tree is **true**. If no proof tree can be produced the value returned is false. We can view the answer **true** returned as a function of the proof tree, all proof trees for the query will return the same answer. Using the calculus of certainties implemented in YAPES, proof trees can return a wider range of values. Using lattices as the domain of certainties allows us to return the whole proof tree as a certainty if so desired. The problem facing the expert system builder using this formulation is to establish the correct degree of abstraction desired from a proof tree. The implementation described above returns the assumptions needed to produce that particular proof tree. It is an important consequence of this approach that the certainty of a query now depends on the manner in which it was proved. This is not the case for standard Horn clause logic.

Let us reconsider the example presented above. We could write rules related to $h$ as follows:

$$f_1\colon h \leftarrow e_1,\ not(e_2). \quad f_2\colon h \leftarrow e_2,\ not(e_1).$$

The goal $not(e_1)$ in the second rule can now be interpreted in terms of failure to find a proof for $e_1$. In the thresholding interpreter used by YAPES the goal $not(G)$ will be true if $G$ does not have a proof with certainty exceeding the threshold. The use of negation in this manner permits the expression of knowledge about conflicting goals, whilst preserving the semantics of the language.

## 3.2. User interaction in YAPES

The user interaction in YAPES is modelled after the 'query the user' principles described in Sergot (1983). The fundamental idea is to consider the user as a data base from which atomic facts necessary to a proof can be obtained. In effect the PROLOG engine has a choice of data bases from which to obtain facts, the 'system' data base, the 'problem domain' data base and the 'user' data base. This idea can also be applied to any data source other than the user, for example on-line data bases.

### 3.2.1. *Categories of question*

There are several different categories of question that can be put to the user. There is some overlap between these categories, but in cases where the overlap occurs there is no conflict in the treatment of the questions.

These are:

(a) Questions which have answer false (e.g. 5 is 2 + 2?);

(b) Ground questions (which have a true/false answer);

(c) Questions which have at most one answer (e.g. sex(tim, *X*));

(d) Questions which may have multiple answers (e.g. is a_son_of(*X*, *Y*)?).

To avoid annoying the user each must be handled in a sensible way. Yapes handles this problem in a way that is intuitively obvious to the user. We shall consider the above categories of question one at a time.

1. Questions which have answer false. These questions are answered by a simple 'no'. It makes no difference whether the question is *ground* (i.e. contains no free variables) or non-ground. The answer to the question is stored in the user data base and will not be asked again, even it the underlying Prolog inference mechanism backtracks to the question.

2. Questions which have answer true or false because they are ground and about which the system has no meta-level information. The user does not need to supply values for variables in the question. These questions are just asked once and the answer stored.

3. Questions which have a unique answer like 'sex(tim, *S*);' are slightly more complicated in that to handle them properly we need to know something about the relation 'sex(*A*, *B*)'. As far as the logic is concerned any given '*A*' could have any number of sexes (just as any sex in fact has many people!). In the case of the 'sex' relation we need to know that sex is a *function* from persons to sexes, this is achieved by the following meta-level declaration to Yapes:

unique(sex(*A*, *B*), [*A*]).

This asserts that if variable '*A*' is instantiated then there is a unique '*B*' that makes 'sex(*A*, *B*)' true. When Yapes needs to find out whether 'sex(tim, male)' is true it deduces from the 'unique' assertion above that the most general question to ask is 'sex(tim, *A*)' (which has a unique answer) and asks this question, suitably formatted. Whenever it subsequently has to solve a goal of the form 'sex(tim, *X*)' it has all the necessary information stored in its data base and need not bother the user.

4. Questions which have multiple answers are also dealt with correctly. Let us consider the relation 'isa_son_of(*X*, *Y*)' intended to mean that *X* has son *Y*, which can have several solutions. A Prolog program using this relation may have to backtrack over several sons before it finds the correct one. Let us consider a specific and rather silly example:

```
chosen_son(Y, X):- %Y  has chosen son  X
   isa_son_of(Y, X),
   age(X, Age>21),
```

A Prolog program will typically find an '*X*' satisfying 'chosen_son(*Y*, *X*)' (with '*Y*' instantiated) by generating possible solutions with 'isa_son_of(*Y*, *X*)' and then the other two goals.

If Yapes has been told that 'isa_son_of(*Y*, *X*)' was in the user data base via the meta-level declaration

askable(isa_son_of(*Y*, *X*)).

if would query the user for instantiations of *X*. When queried the user has three choices:

1. He/she can type in the name of a son. Yapes will then query for another son (since it has not been told anything about the 'isa_son_of' relation).

2. He/she can type 'no', which should be interpreted as meaning that there are no solutions other than those already provided.

3. He/she can type 'enough', which means: enough questions for now, come back if you need any more solutions later. Yapes will come back when and only when it has backtracked over all the previous user answers and none of them can be used to prove the top-level goal.

### 3.2.2. *Tracing and debugging in Yapes*

The user can ask questions of Yapes while a goal is being evaluated by asking 'why' a question has been put. Yapes then displays a trace of the current path on the proof tree to the user, in a suitably formatted manner.

Two other trace/debugging features are currently implemented in Yapes, the 'how' facility and the 'why-not' facility. The 'how' facility enables the user to see *how* a conclusion was reached. Section 4 contains an example of the use of 'how'. Several implementations of *how* are possible. The one currently used in Yapes allows the user to walk through the proof tree of a successful query.

The problem of diagnosing *why* a query failed is in general more difficult. The reason for this is that for a query to fail every possible proof tree must fail. Not only must we ask why a particular goal failed but must take into account the possibility that earlier goals in the tree produced an incorrect answer, rather than failing.

Without information as to the possible failure modes of a program the only practical possibility seems to be interactive debugging with the user as an oracle, providing information as to which goals should not have failed and which goals have produced an incorrect answer.

In an expert system, however, we do have information as to possible failure modes of a program. We can assume that the knowledge-base has been fully debugged by an expert and that the failure of a goal arises from an answer supplied by the user. The 'why-not' facility of Yapes uses this idea.

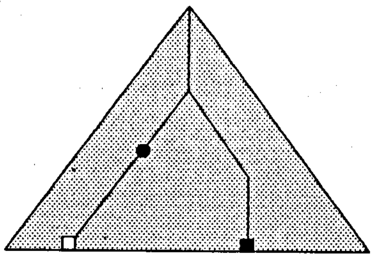


Figure 6. SLD search tree. ■ Node that fails at a user-askable goal. □ Node that fails at a non-user-askable goal. ● Node where a user-askable goal is reduced.

The situation that arises when a goal fails is shown in Figure 6. The SLD-search tree is an *or* tree. Each branch on the tree represents a (partial) proof tree. The tree branches when there is more than one matching clause for a goal. A failing SLD search tree is one where each path through the tree ends in a set of goals that is not satisfiable. In particular if the tree has been produced by PROLOG we know that the leftmost goal is unsatisfiable.

The information that YAPES provides to the user is a failing branch of one of the partial proof trees occurring in the failing SLD-tree. The partial proof tree is selected from a set of candidates in the manner described below, and the branch displayed is that containing the failing goal of the corresponding failed SLD-node.

Candidates are branches of the failed SLD-search tree which:

(a) have as failing goal a user answer; or

(b) have as failing goal a goal which fails to unify because of a substitution occurring at the unification of a user answer with the corresponding atomic goal.

The second condition remainds one of truth-maintenance systems which maintain dependency information about goals (e.g. Doyle, 1979). The difference here is that we are working top-down rather than bottom-up.

There may be many candidates in any given failed SLD-tree. At present YAPES selects a branch that involves the most recent user answer possible. This is a heuristic approach which has worked well so far in practice.

### 3.2.3. *Changing user answers*

Intimately linked with the generation of why-not traces is the ability to change answers to questions. This enables the user to try 'what if' possibilities, and also to correct wrong answers. This facility is currently provided in YAPES.

### 3.3. Type checking user answers

YAPES checks the types of user answers to queries. This facility is based on the PROLOG type checker developed by Mycroft and O'Keefe (1984).

The type checker allows the definition of user-defined types, and the assignment of generic types to procedure arguments.

In practice the user's answer is type checked dynamically and an error message produced if a type violation is detected.

## 4. ANNOTATED TRANSCRIPT OF A YAPES SESSION

The transcript below was taken directly from a terminal session with YAPES. A version of the system with a more sophisticated, window-oriented user interface is also available. All comments are in italics.

*Call YAPES from within PROLOG to start session a knowledge-base has already been loaded.*
*|?-yapes.*

```
Travel allowance evaluation program

    This data base works out whether or not your employee is entitled to
    an allowance after a trip of duty travelling by car.

Welcome to YAPES

Type help for help

Command: h

HELP INFO
  help—This message
  go—Go call the knowledge-base
  break—Break to PROLOG
  change—Change previously recorded facts
  clear—Clear the database of all recorded facts
  why—Why did a query succeed or fail?
  quit—Regretfully leave YAPES

Command: go

What is the name of the employee? tim

Who is the owner of the car? tim
```

Who gave permission for this trip (mse, da, none)? mse

How many kilometres were driven in total? why

*We are asking why this question has been asked. The answer reveals the goals that are currently being investigated.*

```
I'm investigating 'The distance travelled was A km.' to establish:
⟨1⟩  The trip was within holland if
       The distance travelled was A km. and
       A < 300.

⟨2⟩  tim had permission to make this trip if
       Permission for the trip was given by mse and
       Authorization by mse is sufficient for trips to holland and
       The trip was within holland.

⟨3⟩  tim is entitled to an allowance of fl.A if
       The car is owned by tim and
       tim had permission to make this trip and
       tim was properly insured and
       The distance travelled was B km. and
       The uncorrected allowance for a distance of B kilometres is fl.C
       and
       The boat allowance is fl.D and
       The multiplication factor for joint travel is E and
       F is E * (C + D) and
       A plane trip would cost fl.G and
       The maximum of G and F is A.
```

*Now back to the question*

How many kilometres were driven in total? 100

Does tim have a third-party risk insurance for his car? n

*This is the answer to the query*
tim isnt entitled to a travel allowance

*Now we investigate the failure of the goal*
Command: why

```
(e.g. why did the goal fail)
Since
You (the user) stated/implied that:
  tim doesnt have third party risk insurance.

Therefore
  tim wasnt properly insured

Therefore
  tim isnt entitled to a travel allowance
```

*We decide to change a previous answer*

```
Command: ch

You have told me that:

⟨1⟩  The car is owned by tim
⟨2⟩  The employee's name is tim
⟨3⟩  The distance travelled was 100 km.
⟨4⟩  Permission for the trip was given by mse
⟨5⟩  tim doesnt have third-party risk insurance
```

*We change two facts (by deleting them from the data base)*

```
Which facts to be deleted ⟨1.5⟩? 3 5

  Erased

Command: go
```

*Here is the type checker in operation*

```
How many kilometres were driven in total? rty

Sorry incorrect type(s) in your answer
The type(s) of The distance travelled was A km. should be:
  integer

How many kilometres were driven in total? 100

Does tim have a third-party risk insurance for his car? y

Who has been driving during the trip? tim

What is the list price of the car? 10000
```

```
Did tim travel with colleagues? n
```

*Now we succeed with the following assumptions made*

```
Assuming:
⟨1⟩  The trip was in Holland
⟨2⟩  tim didnt go on a boat
⟨3⟩  The plane cost is more than the car cost in Holland
⟨4⟩  The trip was in Holland

then

tim is entitled to an allowance of fl.53
Are any of these assumptions incorrect? y
```

*We don't like the assumption that the trip was not abroad*

```
Give me the list of incorrect assumptions: 1

Where did the trip take place (Holland or abroad)? abroad

tim isnt entitled to a travel allowance

Command: why

(e.g. why did the goal fail)

Since
You (the user) stated/implied that:
  not The trip was within holland.

Therefore
  tim didnt have permission to make this trip

Therefore
  tim isnt entitled to a travel allowance
```

*We change another fact, this time about who gave permission for the trip*

```
Command: ch

You have told me that:

⟨1⟩  The list price of the car is 10000
⟨2⟩  The car is owned by tim
⟨3⟩  tim was driving
⟨4⟩  The employee's name is tim
```

```
⟨5⟩ The trip was within abroad
⟨6⟩ The distance travelled was 100 km.
⟨7⟩ Permission for the trip was given by mse
⟨8⟩ tim has third party risk insurance
⟨9⟩ tim didnt travel with colleagues

Which facts to be deleted ⟨1 · · · 9⟩?7

  Erased

Command: go

Who gave permission for this trip (mse, da, none)? da

How much would an airplane ticket have cost for this trip? 200

Assuming:
⟨1⟩ tim didnt go on a boat

then

tim is entitled to an allowance of fl.200

Are any of these assumptions incorrect? n
```

*Now we are satisfied with the answer we can investigate the solution.*

```
Command: why
(e.g. why did the goal succeed)
tim is entitled to an allowance of fl.200 since
 ⟨1⟩ The car is owned by tim
 ⟨2⟩ tim had permission to make this trip
 ⟨3⟩ tim was properly insured
 ⟨4⟩ The distance travelled was 100 km.
 ⟨5⟩ The uncorrected allowance for a distance of 100 kilometres if fl.52
 ⟨6⟩ The boat allowance is fl.0
 ⟨7⟩ The multiplication factor for joint travel is 1
 ⟨8⟩ 52 is 1 * (52 + 0)
 ⟨9⟩ A plane trip would cost fl.200
⟨10⟩ The maximum of 200 and 52 is 200
```

*We can now investigate one of the subgoals*

```
Which goal (1 to 10)? 2
```

```
tim had permission to make this trip since
⟨1⟩  Permission for the trip was given by da
⟨2⟩  Authorization by da is sufficient for trips to abroad
⟨3⟩  The trip was within abroad

Which goal (1 to 3)? 2
```

*There is a unit clause in the knowledge-base asserting that da can authorise trips abroad*

```
Authorization by da is sufficient for trips to abroad is a fact

Repeat (r) or stop (s) (tim had permission to make this trip)? s

Repeat (r) or stop (s) (tim is entitled to an allowance of fl.200)? r
tim is entitled to an allowance of fl.200 since
 ⟨1⟩  The car is owned by tim
 ⟨2⟩  tim had permission to make this trip
 ⟨3⟩  tim was properly insured
 ⟨4⟩  The distance travelled was 100 km.
 ⟨5⟩  The uncorrected allowance for a distance of 100 kilometres is fl.52
 ⟨6⟩  The boat allowance is fl.0.
 ⟨7⟩  The multiplication factor for joint travel is 1
 ⟨8⟩  52 is 1 * (52 + 0)
 ⟨9⟩  A plane trip would cost fl.200
⟨10⟩  The maximum of 200 and 52 is 200

Which goal (1 to 10)? 1
```

*This is a user-declared fact*

```
I was told that The car is owned by tim is true

Repeat (r) or stop (s) (tim is entitled to an allowance of fl.200)? s
```

*Now we quit YAPES*

```
Command: q

Goodbye—see you again I hope
```

xxxxxxxxxx

## 5. CONCLUSIONS

YAPES runs under C-PROLOG and Quintus PROLOG on VAX and SUN machines. It is currently used as a teaching tool to demonstrate the use of logic programming for expert system design. It has demonstrated that:

1. It is feasible to implement the inference engine of an expert system shell in PROLOG. Using Horn clause logic as the knowledge representation language provides a smooth interface with the underlying PROLOG.

2. Explanation facilities can be based on the proof generated by PROLOG as it solves a goal. This proof does not depend on the execution strategy of the interpreter, so that explanation facilities are unaffected by the use of different control regimes. Considering the user as a separate data base of facts allows us to diagnose failure of goals (why not questions) effectively without further interaction with the user.

3. It is useful to interpret plausible inference as an extension to the language of Horn clause logic. In particular we can view plausible inference as concerning the manner in which a goal is proved rather than its truth value *per se*. Further work is necessary to investigate ways in which different proofs of a goal can be manipulated and presented to the user in an informative manner.

**Acknowledgements**

This research was conducted in cooperation with the Shell Research Centre at Sittingbourne Kent, to whom thanks are due for support of the work described and for permission to publish this note.

## APPENDIX A: LOGIC PROGRAMMING WITH UNCERTAINTIES

### 1. Introduction

A recent paper by Shapiro (1983c) introduces the idea of adding uncertainties to logic programs. This idea has been suggested previously: for example Scott and Krauss (1966) develop the semantics of probabilistic logic for countable and infinitary logics. Shapiro's approach is of interest because it provides a simple semantics for Horn clause programs with uncertainties, and is more general than previous approaches—providing a richer set of combining functions for certainties.

This note describes Shapiro's approach and generalizes his results in a number of ways. In particular a fixpoint semantics is given, more general classes of uncertainty functions are allowed and the domain of certainties is extended.

This approach is of particular interest to those involved in the development of expert systems. A large class of such systems use one form or other of uncertain reasoning. Few of these systems are well founded from a theoretical point of view. Logic programs with uncertainties may provide a more suitable framework in which to pursue uncertain reasoning.

## 2. Basic definitions

*Definition.* A certainty space **C** is a complete lattice. This is a partially ordered (under $\leq$) set $C$, with operations $\cup$ and $\cap$ (least upper bound and greatest lower bound respectively) defined for every subset of $C$.
*Definition.* We extend $\leq$ to sequences over $C$ by defining: $\langle c_1, \ldots, c_n \rangle \leq \langle c'_1, \ldots, c'_n \rangle$ iff $c_1 \leq c'_1$ and ... and $c_n \leq c'_n$.
*Definition.* A function **f** from sequences of certainties to certainties in some certainty space **C** is monotone iff for all sequences $s_1$ and $s_2$ of length $n$ over **C**, $s_1 \leq s_2$ implies $f(s_1) \leq f(s_2)$.
*Definition.* A logic program with uncertainties is a finite (non-empty) set **P** of pairs of the form $\langle \mathbf{A} \leftarrow \mathbf{B}, \mathbf{f} \rangle$ where $\mathbf{A} \leftarrow \mathbf{B}$ is a definite (or Horn) clause, and **f** is a montone function from sequences of certainties to certainties.

## 3. Semantics

We are now in a position to define a semantics for logic programs with uncertainties. This semantics is an extension to that given by Van Emden and Kowalski (1976) for definite clause programs without uncertainties. The reader can check that the semantics given here reduces to the semantics of logic programs without uncertainties in the case that our domain of certainties is {**false, true**} together with appropriate definitions of the functions **f**.
*Definition.* The Herbrand Universe **U(P)** is defined recursively as,

1. The set of constant symbols in **P** (or the constant symbol **a** if there are none).
2. All atoms of the form $\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ where the $\mathbf{t}_i$ are in **U(P)**.

*Definition.* The Herbrand base **H(P)** of a logic program **P** is the set of all ground atoms formed by using predicate symbols from **P** with ground terms from the Herbrand Universe **U(P)**.
*Definition.* An interpretation of a logic program with uncertainties **P** is a function from **H(P)** to a certainty space **C**.
*Definition.* An interpretation $\mathbf{I}_1$ is $\leq$ an interpretation $\mathbf{I}_2$ iff $\mathbf{I}_1(\mathbf{a}) \leq \mathbf{I}_2(\mathbf{a})$ for all $a$ in **H(P)**.
*Definition.* A model **M** of **P** is an interpretation of **P** satisfying the following condition: for any clasue $\langle A \leftarrow B, f \rangle$ in **P** and any ground instance $A' \leftarrow B'_1 \;\&\cdots\&\; B'_n$ of the clause, then $\mathbf{M}(A') \geq f(\mathbf{M}(B'_1), \ldots, \mathbf{M}(B'_n))$.

### 3.1. *Model theoretic semantics*

*Definition.* Given two models $M_1$ and $M_2$ for **P** we define the intersection ($\cap$) of the two models pointwise as $M_1 \cap M_2(a) = M_1(a) \cap M_2(a)$
*Proposition.* Given two models $M$ and $M'$ for **P**, the intersection of the two models $M \cap M'$ is a model.

*Proof.* For any $a \in \mathbf{H}(\mathbf{P})$ if there is a ground instance $a \leftarrow b_1 \ \& \cdots \& \ b_n$ of a clause in $\mathbf{P}$, then the following conditions hold:

$$\begin{aligned} \text{In } M\colon M(a) &\geq f(\langle M(b_1), \ldots, M(b_n)\rangle) \\ &\geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle) \\ \text{In } M'\colon M'(a) &\geq f(\langle M'(b_1), \ldots, M'(b_n)\rangle) \\ &\geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle). \\ \text{As } \mathbf{C} \text{ is a lattice } M \cap M'(a) &= M(a) \cap M'(a) \\ &\geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle) \end{aligned}$$

which was to be proved.

*Theorem.* $\mathbf{M}(\mathbf{P}) = \bigcap_{M \text{ a model}} M$ exists and is the least model of $\mathbf{P}$.

*Proof.* The proof is straightforward.

The meaning of the logic program $\mathbf{P}$ is defined to be the least model $\mathbf{M}(\mathbf{P})$ of $\mathbf{P}$

## 3.2. *Fixpoint semantics*

We now provide a constructive definition of $\mathbf{M}(\mathbf{P})$ by showing it to be the least fixpoint of a function $\mathbf{T_P}$ from interpretations to interpretations.

*Definition.* Let $\mathbf{T_P}(I)$ be defined pointwise on an interpretation $I$ as follows:

$\mathbf{T_P}(I)(a) = \bigcup_{\lambda \in \Lambda_a} (f_\lambda(I))$ where $\Lambda_a = \{a \leftarrow b \mid a \leftarrow b \textit{ is a ground instance of a clause in } \mathbf{P}\}$ and $f_\lambda(I) = f(I(b_1), \ldots, I(b_n))$ where $\lambda = a \leftarrow b_1 \ \& . \cdots \& \ b_n$.

*Proposition.* $\mathbf{T_P}$ is monotone over the lattice of interpretations, and moreover the lattice of interpretations (with *po*$\leq$) is complete.

*Proposition.* $\mathbf{T_P}$ has a least fixpoint.

*Proof.* Since $\mathbf{T_P}$ is a monotone function on a complete lattice, $\mathbf{T_P}$ has a complete lattice of fixpoints and therefore a least fixpoint.

We can elaborate on this result by proving that $\mathbf{T_P}$ is **continuous** as well as monotone if the underlying certainty functions we use are continuous.

*Definition.* A certain function $\mathbf{f}$ from sequences of certainties to certainties is continuous iff $\mathbf{f}(\bigcup_{x \in X} x) = \bigcup_{x \in X} \mathbf{f}(x)$ for all chains $X$.

*Proposition.* If the underlying cerainty functions $\mathbf{f}$ are continuous then the corresponding functions $\mathbf{f}_\lambda$ are continuous.

*Proposition.* If the underlying certainty functions $\mathbf{f}$ are continuous then $\mathbf{T_P}$ is a continuous function, that is: $\bigcup_{n \in \Pi} \mathbf{T}_P(I_n) = \mathbf{T}_P(\bigcup_{n \in \Pi} I_n)$ for all chains $\{I_n \mid n \in \Pi\}$ of interpretations.

*Proof.*

$$\begin{aligned}
\mathbf{T_P}\Big(\bigcup_n I_n\Big)(a) &= \bigcup_{\lambda\in\Lambda_a} f_\lambda\Big(\bigcup_n I_n\Big) \quad \text{(by definition of } \mathbf{T_P}\text{)}\\
&= \bigcup_\lambda \bigcup_n f_\lambda(I_n) \quad \text{(by continuity of the } f_\lambda\text{)}\\
&= \bigcup_n \bigcup_\lambda f_\lambda(I_n) \quad \text{(by interchangeability of } lubs\text{)}\\
&= \bigcup_n \mathbf{T_P}(I_n)(a) \quad \text{(by definition of } \mathbf{T_P}\text{)}
\end{aligned}$$

This completes our discussion of the semantics of logic programs with uncertainties, and shows that the meaning of a logic program with uncertainties can be approximated computationally.

## 4. Discussion

### 4.1. *Relation to Shapiro's work*

We have extended the results presented by Shapiro in three ways.

1. The semantics has been extended to provide a fixpoint semantics for uncertainties. It has been shown that if the underlying certainty functions are continuous the certainties of formulas can be approximated computationally.

2. The class of functions used for evaluating certainties is broader, as they use sequences of certainties as the domain rather than multisets of certainties as used by Shapiro. This allows one to distinguish a given literal in a clause as being more important (less important) than others, which is often the case in practical reasoning.

3. The certainty space used is a complete lattice, rather than the interval (0, 1] used by Shapiro. This is important because it permits us to use uncertainties which are not necessarily 'comparable', by insisting only that certainties are partially ordered rather than totally ordered. This is important in practical reasoning, where a total ordering of certainties often requires an expert to make problematic judgements, unwarranted by experience.

## APPENDIX B: THE TOP LEVEL INTERPRETER FOR YAPES

```
% solve/4 has arguments:
%   *Goal*—Goal to be solved
%   *Trace*—Current goal stack
%   *Thresh*—Threshold value
%   *Value*—the assumptions under which the goal holds solve
```

```
solve ((A, B), Trace, Thresh, Value)
  :- !,
     solve(A, Trace, Thresh, Aval),
     solve(B, Trace, Thresh, Bval),
     combine_and(Aval, Bval, Value),  % Combine assumptions for
     ∵d combo
     less(Thresh, Value).  % Value is 'not less' than *Thresh*
                           %the threshold value
% A→B; C is Prolog's form of if · · · then · · · then · · · else
solve ((A→B; C), Trace, Thresh, Value)
  :- !,
     (solve(A, Trace, Thresh, Aval), !,
     solve(B, Trace, Thresh, Bval),
     combine_and(Aval,Bval, Value)
     ;
     solve(C, Trace, Thresh, Value)
     ).
solve((A; B), Trace, Thresh, Value)
  :- !,
     C
     solve(A, Trace, Thresh, Value)
     ;
     solve(B, Trace, Thresh, Value)
     ).
solve(not Goal, Trace, Thresh, [])
  - !, % This cut is essential.
     not solve(Goal, Trace, Thresh,_).
% System goals are always solved without assumptions (they're correct!)
solve(Goal, Trace, Thresh, [])
  :- system(Goal)
     !,
     call(Goal).

% Reduce the goal *Goal* and incorporate its assumptions
% The reduction is by access to non-system clauses,
% either unit clauses provided interactively by the user or clauses in the
% rulebase provided by the user.
solve(Goal, Trace, Thresh, Value)
  :- dataclause(Just, Goal, Body, Trace),
    solve(Body, [(Just:Goal:- body) | Trace], Thresh, BodyVal),
    combine_if(BodyVal, Just, Value),
    less(Thresh, Value).
```

## REFERENCES

Cheeseman, P. (1983) A method of computing generalised Bayesian probability values for expert systems. *IJCAI-83,* 198–202.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence* **12**(3), 231–272.

Duda, R. *et al.* (1979) Model design in the PROSPECTOR consultant system for mineral exploration. In *Expert systems in the micro-electronic age* (ed. D. Michie) pp. 153–167. Edinburgh University Press, Edinburgh.

Hammond, P. (1983) *APES: A Prolog expert system shell,* Department of Computing, Imperial College, London.

Kowalski, R. (1979) *Logic for problem solving.* North Holland, Amsterdam.

Mycroft, A and O'Keefe R. (1984) A polymorphic type system for Prolog. *Artificial Intelligence* **23**(3) 153–167.

Scott, D. and Krauss, P. (1966) Assigning probabilities to logical formulae. In *Aspects of inductive logic* (eds E. Hintikka and P. Suppes) pp. 219–264. North Holland, Amsterdam.

Sergot, M. (1983) A query-the-user facility for logic programming, *Proc. European Conf. on Integrated Interactive Computing Systems,* (eds P. Degano and E. Sandewall), North Holland, Amsterdam.

Shapiro, E. H. (1983a) *A subset of concurrent Prolog and its interpreter.* TR-003. ICOT, Tokyo.

Shapiro, E. H. (1983b) *Algorithmic program debugging.* MIT Press, Cambridge, Mass.

Shapiro, E. Y. (1983c) Logic programs with uncertainties. *IJCAI-83*, 529–532.

Shortliffe, E. H. (1976) *Computer based medical consultations: MYCIN.* Elsevier, Amsterdam.

Van Emden, M. H. and Kowalski, R. A. (1976) The semantics of predicate logic as a programming language. *J. Association for Computing Machinery* **23,** 733–742.

Weyhrauch, R. (1980) Prologomena to a theory of mechanized formal reasoning. *Artificial Intelligence* **13,** 133–70.

9

# LogiCalc: a PROLOG Spreadsheet

F. Kriwaczek
Department of Computing,
Imperial College of Science and Technology
London, UK

**Abstract**

LogiCalc is the outcome of an attempt to reconstruct logically, in micro-PROLOG, a typical spreadsheet program. The main features of a spreadsheet can be expressed in a straightforward fashion in logic, and, in the process, the underlying concepts involved can become more clearly exposed. As a consequence, ways of generalizing aspects of conventional spreadsheets emerge naturally. For example, in LogiCalc the functional relationship between cells need not be numeric. Indeed, one can constrain the values of sets of cells so that they jointly satisfy non-functional relationships. As well as taking string or numeric values, cells of LogiCalc can be lists, diagrams, pieces of text, or even spreadsheets, which bear a relation to other cells. Items such as these can be viewed through windows onto the cells. LogiCalc could be incorporated within front ends for expert systems, or form the basis for more intelligent decision support tools.

## 1. INTRODUCTION

The spreadsheet or worksheet is a device employed by accountants to lay out a set of figures in an organized fashion. It consists simply of a large sheet of paper divided into a rectangular array of cells by a grid of horizontal and vertical lines. Each cell can hold a string constant, such as the name of a salesman or a product, or, alternatively, a numeric constant, such as the total revenue achieved by a particular division over the last financial year or the forecast rate of increase of sales over the next five years.

Normally a variety of functional relationships hold between the items in different cells. For example, the figure for profits would usually be equal to the difference between the figure for revenue and the figure for costs. When the figure in a cell changes, either because of an earlier error or because it was based on an assumption that has since been modified, all the figures that depend on this cell have to be erased and new values calculated by hand.

The first electronic spreadsheet, VisiCalc, was developed by Bricklin, Frankston and Fylstra in 1978. It was designed to avoid the labour involved in recalculation. The display resembles a paper spreadsheet, but unlike the latter, a cell of VisiCalc can also hold a mathematical formula. What is displayed in such a cell is not the explicit formula, but the numeric value that results from applying the formula. In VisiCalc, if any item in a cell is changed, all the items that depend on this value are recalculated automatically and the new values displayed.

Using an electronic spreadsheet it is possible to build small to medium mathematical models. Cells with numeric constants hold the values of the exogenous variables, whilst mathematical formulas represent the model's structure. By experimenting with different values for exogenous variables it is easy to simulate the consequences of different decisions or of different assumptions about the state of the world—the "What if?" approach.

The immense commercial success of VisiCalc and other spreadsheet programs which have been inspired by it, such as Lotus 1–2–3, attests to the power and flexibility of the electronic spreadsheet. Furthermore, it seems to indicate that in many real-life situations naïve computer users find it perfectly natural and indeed often preferable to express their problems in a descriptive rather than an algorithmic form. The spreadsheet can be regarded as the most popular of all declarative languages, perhaps as popular as the procedural language BASIC.

This paper describes LogiCalc, an implementation of a spreadsheet program in the logic programming language micro-PROLOG. It was initially developed as part of the author's M.Sc. project [1], and bears no relation to a commercially distributed spreadsheet program of the same name. The first part of this paper describes the main aspects of LogiCalc's behaviour, explaining how the items in each cell are held internally, and how the spreadsheet is displayed. Next comes a description of those features of LogiCalc that are not normally found in spreadsheet programs. Many of these follow naturally from generalizing concepts found in conventional spreadsheets. Finally, there is a brief discussion about potential areas for development.

The syntax used in the examples of PROLOG code is that of the SIMPLE front-end to micro-PROLOG [2]. Logical variables are denoted by upper- or lower-case *X*, *Y* or *Z*, possibly followed by an integer. In this paper, string constant argument values are written between quotation marks, to distinguish them from predicate names.

## 2. LOGICALC'S GENERAL BEHAVIOUR

As is common practice, the columns of LogiCalc are labelled by letters and the rows by numbers. Each cell is normally referred to by its coordinate position, with *A*1 being the top left-hand cell, although it also possible to declare meaningful synonyms for individual cells and ranges of cells. There are no formal restrictions on the spreadsheet's dimensions, but in practice the number of non-empty cells is limited by the core size of the computer. At any one time only a small part of the spreadsheet is visible. The screen includes a window showing a rectangular portion of the spreadsheet, normally containing 20 rows and between one and nine columns (depending on the chosen column width). The PROLOG workspace holds a "window-specification" assertion giving details of the position in the spreadsheet of the current window together with the latter's dimensions.

One cell in the window, the cursor cell, is displayed in reverse video. By using the arrow keys, or a special move command, the cursor can be shifted to other cells. A "cursor-cell" assertion in the workspace holds the coordinates of the current cursor cell. Before defining (or redefining) the contents of a cell, the cursor must be moved to that cell. As well as being able to enter a definition for the cursor cell, the user can also input a variety of commands. There are commands for moving the cursor within the window, moving the whole window, changing the format for output of values, changing the column width, saving the clauses for a complete model onto disc, loading a model from disc, and so on. When the contents of a cell are being defined, the system is normally able to detect automatically whether the item is a string constant, a numeric constant or a formula, and to handle the item appropriately. The system can recognize a command by its special initial character. In most cases this character is "/", and is followed by one or more letters signifying the particular kind of command. User input, whether it be a definition for the cursor cell or a command, usually leads to an update of the PROLOG workspace, followed by a redisplay of the spreadsheet window in accordance with the clauses in this new workspace. For many of the commands, including those for changing the output format of cell values, movement of the window to a different part of the spreadsheet or loading the clauses for a new spreadsheet model from disc, redisplay involves clearing the spreadsheet window and drawing it afresh. On the other hand, redisplay following the input of a cell definition need not be so drastic. At the very least, however, new values must overwrite the old ones for all cells whose values have changed as a consequence. Such considerations were crucial when it came to choosing the way to represent the contents of cells in LogiCalc.

In LogiCalc, when a constant value is entered in a cell, a "has-definition" assertion such as

"$A2$" has-definition "SALES"

or

"$B2$" has-definition 1000

is created.

If a mathematical formula is to be held in a cell then the formula, as entered by the user, is processed by an expression parser akin to that of the SIMPLE front-end of micro-PROLOG [2], and a "has-definition" rule created.

In the first implementation of LogiCalc, if a user had entered the formula $B2*B7$ for cell $C2$, say, then the system would create the rule

```
"C2" has-definition X
  if "B2" has-definition Y
  and "B7" has-definition Z
  and X = Y * Z
```

When producing a revised spreadsheet, the system would, for each cell in the window in turn, calculate its latest value from first principles and then display the values on the screen. This was done in top-down fashion, according to its "has-definition" clause. If no "has-definition" rule was found for a cell, then a blank string would be displayed, by default. Although this arrangement was perfectly straightforward, it tended to be inefficient.

An immediate improvement in efficiency was gained by observing that spreadsheet windows often have many blank cells. In order to avoid explicitly writing blank strings at all these places, an "active-row" assertion is now held in the workspace, storing a list of those rows of the spreadsheet having cells with definitions. For each row occurring in this list an "active-column" assertion stores a list of those positions in the row at which there are cells with definitions. Using the "active-row", active-column and "window-specification" assertions it is easy to find which cells in the window have definitions. Only the current values of these cells need be put on the screen.

However, the above arrangement still involves a great deal of unnecessary calculation. Consider even the very simple example in which cells $A1$, $B1$ and $C1$ are defined by

$$A1 = 100*100$$
$$B1 = 2*A1$$
$$C1 = A1 + B1$$

If, in displaying a spreadsheet window, the values for $A1$, $B1$ and $C1$

have to be calculated from basic principles, then the value for *A*1 must be computed three times and the value for *B*1 twice.

By keeping in the workspace an explicit record of the current value of each active spreadsheet cell, together with information about the dependency between cells, much superfluous effort can be avoided. When a cell receives a definition (or redefinition), the values of precisely those cells affected will be recalculated. In these circumstances, it is no longer necessary to redisplay the values of all cells in the window, but only those in the window whose values have been recalculated. On the other hand, if a window has to be drawn afresh following a command, then the values of all active cells can be accessed directly, rather than having to be calculated from scratch.

In the present implementation of LogiCalc, if a user enters a string or numeric constant in a cell, then, in addition to a "has-definition" assertion, as described above, a corresponding "has-value" assertion such as

"*A*2" has-value "SALES"

or

"*B*2" has-value 1000

would also be created.

The form of the "has-definition" rule for cells holding a formula differs from that of the original implementation. Keeping to the example of cell *C*2, defined by the formula $B2*B7$, the "has-definition"' clause for this cell would, in the latest implementation of LogiCalc, be

```
"C2" has-definition X
  if "B2" has-value Y
  and "B7" has-value Z
  and X = Y * Z
```

As soon as a "has-definition" rule such as this is created, a "has-definition" query is run for that cell, and the resulting explicit value stored in a "has-value" assertion. This means that, in the above case, provided *B*2 and *B*7 have already been defined and their current values stored in "has-value" clauses, the job of evaluating *C*2 and creating a "has-value" assertion for this cell is quite direct. Were one or both of *B*2 and *B*7 not yet defined, *C*2 would temporarily be given an ERROR value.

It is the "has-value" assertion that the system accesses when displaying the spreadsheet window. When any item changes, the "has-value" clauses for all cells dependent upon this item have to be updated immediately. So that this may be done efficiently the dependency graph between the cells of the spreadsheet is held in the workspace in the form of "dependent-upon" assertions. In the present example, it can be seen that cell *C*2 is directly dependent upon cells *B*2 and *B*7, and the following clauses would be created automatically:

"*C*2" dependent-upon "*B*2"
"*C*2" dependent-upon "*B*7"

Were the "has-definition" clause for *C*2 to be replaced by a new one, then first of all those "dependent-upon" assertions which give the cells upon which *C*2 itself was directly dependent, under its former definition, would be deleted. Next a new set of "dependent-upon" clauses would be created, which give the cells upon which *C*2 iteslf is now directly dependent. Finally, all those cells that are descendants of *C*2 (in the sense that they are either directly dependent upon *C*2, or are directly dependent upon descendants of *C*2) would have their new values calculated using the "has-definition" rules, and their "has-value" assertions updated.

It is important to ensure that this updating of descendant cell values is done in a sensible order. Suppose, for example, that the following definitions hold

$J1 = 100$
$J2 = J1 + J3$
$J3 = 2 * J1$

and that the contents of *J*1, *J*2, and *J*3 are currently correctly evaluated as 100, 300, and 200, respectively. If *J*1 is redefined as 1000 and the values of dependent cells *J*2 and *J*3 are recalculated in that order, then *J*2 will be given the wrong value 1200. The "has-value" clause of *J*3 should have been updated before that of *J*2, and if that had been done then *J*2 would have been given the correct value 3000.

The user can choose between two alternative modes of recalculation. Either the system topologically sorts the set of descendant cells, which are partially ordered by the "dependent-upon" relation, into a consistent total ordering [3]. Or the system sorts the set of descendant cells according to spreadsheet order, (with cell *X* coming before cell *Y* if *X* is in a n earlier row or in the same row but to the left). The latter method is faster, but assumes that the user has laid out his model in such a way that all cells dependent on a given cell are either on the same row and to the right or on a lower row than the given cell.

In either case, the updating of values and their display for cells within the window is co-routined with the sorting, so that the user does not experience a long wait before the screen is updated. Just before displaying the values of cells the system also accesses assertions that give the output format, i.e. number of decimal places for numbers and whether values should be left or right justified.

One of the most powerful features of an electronic spreadsheet is replication. This enables a definition entered for a particular cell to be repeated over a range of cells, across or down the spreadsheet. A formula can either remain unchanged or can be generalized to maintain the patterns between the position of the cell whose contents are being defined and the positions of the cells referenced in the formula.

In LogiCalc replication involves analysing the "has-definition" clause of the original cell and producing a "has-definition" general rule which applies to the complete range of cells across which the definition is to be replicated.

Suppose that the user has entered the command that the definition for cell *C*2 is to be replicated from *D*2 to *G*2. Recall that the "has-definition" rule for *C*2 is

```
"C2" has-definition X
  if "B2" has-value Y
  and "B7" has-value Z
  and X = Y * Z
```

For each cell named in a "has-value" condition in the clause (in this case *B*2 and *B*7) the user is asked whether the cell coordinate should be taken as being relative to the original cell or absolute. Assume that the user wishes *B*2 to be relative, but *B*7 to be absolute, so that *D*2 would be defined as *C*2 * *B*7, *E*2 would be defined as *D*2 * *B*7, and so on.

The system finds the row and column displacement of each relative cell reference from the original cell and uses these to create a new generalized "has-definition" rule. In the present example, the "has-definition" rule created for cells *D*2–*G*2 would be

```
X has-definition X
  if x belongs-to ("D2" "E2" "F2" "G2")
  and displaced (x − 1 0 y)
  and y has-value Y
  and B7 has-value Z
  and X = Y * Z
```

where condition "displaced (*x* − 1 0 *y*)" finds the cell *y* that has column displacement −1 and row displacement 0 with respect to cell *x* (i.e. it lies in the previous column and the same row).

Whenever a cell receives a definition, either directly or through replication, the system looks to see whether there already is a "has-definition" clause for that cell. It if finds a "has-definition" clause specifically for that cell, then it replaces this clause by the new one. If it finds a general "has-definition" rule that applies to the cell in question, then it will ensure that this rule no longer applies by removing the coordinates of that cell from the list in the "belongs-to" condition of the rule, before adding the new "has-definition" clause. If the resulting list in the "belongs-to" condition is empty, then the general "has-definition" rule no longer applies to any cell and is removed from the workspace.

Generalized "dependent-upon" clauses are also created in the course of replication. However, for each cell covered by a generalized "has-definition" clause there will be an individual "has-value" assertion.

## 3. SPECIAL FEATURES OF LOGICALC

### 3.1. Accessing a knowledge base

In addition to holding the current state of the LogiCalc spreadsheet, the PROLOG workspace can also hold another data base in the form of sets of assertions, or, more generally, sets of clauses representing a particular domain of knowledge.

Perhaps the simplest way of accessing a knowledge base from within LogiCalc is by requesting that the extension of a particular relation of interest be inserted, in tabular form, at a given part of the spreadsheet. This not only allows the tuples of the relation to be inspected, but also makes the values available as an integral part of the spreadsheet's model.

Conversely, if a set of tuples has been entered by hand into a rectangle of the spreadsheet, the corresponding relation can be saved as a set of PROLOG assertions. In addition to providing a convenient form of data entry, this arrangement allows the relation so created to be referenced subsequently from within the definitions of the other cells. For example, a $12 \times 2$ rectangle of values giving the number of days in each month could be entered by the user and then saved as the "days-in-month" relation. As we shall see below, such a relation could then be used when defining other cells.

A second way in which a knowledge base can be accessed is through extending the permissible kinds of formulas by which a cell can be defined. As with many conventional spreadsheets, LogiCalc has certain built in mathematical functions, including "@SUM" for the total numeric value of a range of cells. This notation has been extended to cover relations in the knowledge base for which the value of the first argument is uniquely determined by the values of the remaining arguments.

For example, suppose that the knowledge base contains a set of assertions of the form

"Peter" father-of "Sue"
"John" father-of "Michael"
"John" father-of "Jenny"

and so on. Then formulas for defining cells can include the function "@father-of". Just as with mathematical expressions, complex non-numeric formulae can be built up, so that cell *B*2 might, with the appropriate knowledge base, be defined by

@address-of (@employer-of (*A*2)).

Just as with other definitions, this one could be replicated from *B*3 to *B*50, say, so that if column *A* contained people's names, column *B* would automatically give the addresses of their employers.

When using the SIMPLE front-end to micro-PROLOG the normal method of querying a data base is through a "which" query, which finds all the solutions to a conjunctive condition. For example, the query

which(*x*:"Henry" father-of *x* and *x* male)

would, given an appropriate set of clauses in the workspace, find all the sons of Henry. In LogiCalc a cell can be defined by a "which" query. The cell takes for its value a solution to the query. Once a solution is found it is entered in that cell. The user can then choose between sticking with that value and not seeking further solutions, requesting the system to search for another solution, or temporarily keeping the latest solution, whilst moving to other parts of the spreadsheet to carry out other operations, perhaps related to the value last found, but reserving the right to return to the cell and seek further solutions.

In addition to displaying the latest solution, in the cell with the "which" definition, the user can arrange for all the solutions found to be inserted, one at a time, either down a given column or along a given row of cells. Each of these cells is defined as holding its solution as a constant value.

The syntax of "which" queries within LogiCalc has been extended to allow other cells to be referenced within the condition of the query. For example

which(*x*:*C*4 father-of *x* and *x* male)

finds all the sons of the man currently named in *C*4. A change in the value of cell *C*4 will result in different solutions. Cell references can even stand for relation names within the condition of the query. Thus

which(*x*:*C*4 father-of *x* and *x* *D*3)

relies on *D*3 containing the name of some appropriate unary predicate such as "male", "female", or "logician".

The solution to a "which" query could quite conceivably be a tuple. For example, if a cell were defined by

which((*xyz*):*x* father-of *y* and *y* father-of *z*)

then each solution would be a triple of names bearing the "grandfather–father–child" relation. Thus cell values will not always be individual items.

Even though a cell defined by a "which" query may no longer have a single value, the dependency between that cell and any cell referenced within the query's condition is strictly one way. However, in some circumstances it is useful to be able to declare that a particular relationship holds between the values of a given set of cells, without imposing any kind of functionality or direction of dependency. The system will attempt to solve a declared relationship, consistently with any current values of active cells named in the declaration, and will display the components of the solution in the appropriate cells.

Suppose, for example, that the relationship

*K*11 father-of *L*11

has been declared. If neither cell *K*11 nor cell *L*11 has a definition, then the system will attempt to find a pair of values to satisfy this relation and will display them in the corresponding two cells. If *K*11 has a definition, and a current value "pop", say, but *L*11 has none, then the system will attempt to find a value *x* for which

pop father-of *x*

holds. If it finds such a value it will display it in cell *L*11. The system behaves in an analogous fashion if *L*11 has a definition, but not *K*11. Finally, if both *K*11 and *L*11 have definitions, the system will check that their current values satisfy the "father-of" relation. If they do not, the user will be requested either to retract the declaration or to modify the definition of one or both of the cells.

LogiCalc handles a series of such declarations by attempting to solve the conjunction of the relationships. Thus, if the user had declared

*A*1 father-of *A*2
*A*2 father-of *A*3
*A*3 father-of *A*4
. . .
*A*8 father-of *A*9

and any one of *A*1–*A*9 was defined as someone's name, then, provided the relevant facts were in the workspace, LogiCalc would fill in the remainder of cells *A*1–*A*9 with that person's father, grandfather, great-grandfather, etc., and son, grandson, great-grandson, etc.

When the system finds a solution to any declared relationship(s), it displays the solution component values in the corresponding cells, creates "has-value" assertions for these cells, and then recalculates any descendant cell values. The user has the option of either sticking with the current solution or seeking another solution. When the user decides to stick, or the system cannot find another solution, the user can choose between erasing the solution component values from their cells, in which case their "has-value" assertions are deleted, or of defining these cells as holding their components as constant values.

The pre-existence of defined values in cells referred to in a declared relationship can be thought of as adding additional (positive) constraints to the goal of solving the relationship. Negative constraints can be imposed by the user through "forbid" commands. A "forbid" command ensures that a chosen cell will not take a particular value or values.

With the ability to add extra positive or negative constraints and to relax them at will, LogiCalc can be employed to solve scheduling problems, such as timetabling, in an interactive fashion. A relatively simple version of the problem is first posed, leaving out the fine detail. The problem is expressed as a declaration that a particular array of cells are related in such a way that they form the components of the solution. After the system has found a feasible solution to the initial problem, the user can comment on aspects of the solution. These comments can either be positive—insisting that a particular component retains its current value in future, (by defining the cell in question to take that constant value), or negative—forbidding a cell to take the current value in future. Insisting and forbidding can subsequently be relaxed. The system then searches for a new solution, with the extra constraints in force, and, having found a solution, the user can again comment. The process is repeated until the user is satisfied with a solution found. The surface syntax of LogiCalc makes it relatively straightforward to navigate interactively around a search space of solutions.

If the knowledge base to be accessed is large, space considerations may dictate that much or all of the knowledge base be kept on disc rather than in core. LogiCalc makes use of the EXREL utility of micro-PROLOG [2], which allows the system to access clauses from disc files in a transparent way.

Since LogiCalc is essentially interactive in use, it seems unreasonable to expect that all facts required to find the current value for a cell be held in advance in core or on disc. In accordance with Sergot's "Query the User" declarative model of interactive logic programming [4], the user is regarded as an extension of the computer's own knowledge base, and any information not present in core or on a disc file is assumed to be available from the user and is requested from him.

In particular, it is possible to define a cell as taking a value that is 'askable' from the user. When making such a definition the user will be requested to enter an appropriate natural language template. For example, in a spreadsheet model dealing with income tax, cell *F*5 might be defined as holding the number of children of *F*3, askable from the user. The question template might be "How many children does *F*3 have?". Whenever this question was actually posed by the computer, the cell reference, *F*3, would be replaced by that cell's current value.

With its variety of facilities for accessing knowledge bases, LogiCalc could be employed in suitably tailored form as a front end for expert systems, particularly those dealing with financial or commercial domains.

### 3.2. Target reaching

Although spreadsheet models are normally employed in "What if?" investigations, it is often useful to reverse the process and ask what changes in the basic assumptions of the model would have to be made in order to bring about a specified target result.

We have explained that by declaring general relationships amongst sets of cells the directionality of dependency between cells, implicit in functional definitions, is avoided. As a consequence, models that involve such declarations are potentially more flexible in use e.g. target reaching could be achieved by using declared relations rather than functions.

In simple cases this is certainly possible. Consider, for example, the relation "TIMES (*X Y Z*)" of micro-PROLOG, meaning that $X * Y = Z$. The built-in predicate "TIMES" can be used to multiply, divide, or check a product. It calls on machine-code floating-point multiplication and division routines, rather than a vast multiplication table, and in consequence at least two of its arguments must be given. If a user stores the current £–$ conversion rate in cell *A*1 and declares the relationship

TIMES (*A*1 *B*1 *C*1)

then, provided he enters a number of pounds sterling in cell *B*1, the system will display the corresponding number of dollars in cell *C*1. Conversely, if *C*1 holds a number of dollars, then the corresponding number of pounds will be displayed in cell *B*1.

However, mathematical relationships do not always work multidirectionally in PROLOG, due to the way that arithmetical primitives are implemented. The micro-PROLOG query

? TIMES (*X Y* 12)

where *X* and *Y* are uninstantiated variables, would result in a control error. Thus, one could not hope to find a square root directly with a query such as

? TIMES (*X X* 2)

let alone expect to solve quintic equations expressed in terms of such arithmetical primitive relations.

Accordingly, target reaching with complicated models involving arithmetic is implemented in LogiCalc procedurally, using the unsophisticated but normally effective numerical "Method of False Position". The user specifies a target value for the current cursor cell, together with the coordinates of an "assumption" cell whose value is to be found. He also provides two initial estimated values for the "assumption" cell, on the low and high side. The system identifies the subgraph of the dependency graph lying between the "assumption" cell and the "target" cell, and sorts its nodes into a consistent total ordering. This results in a chain of cells between the "assumption" and "target" cells. Any tentative value given to the "assumption" cell will cause a ripple of updated values along this chain, with the "target" cell finally receiving its new value.

The system checks that when applying the user's two estimates, *A* and *B*, to the "assumption" cell, the two resulting values, *C* and *D*, say, for the "target" cell lie on either side of the target value, *T*. The system then applies inverse linear interpolation to obtain a third estimate, *E*, for the "assumption" cell's value, lying somewhere between *A* and *B*. Next, the system determines the value, *F*, of the "target" cell resulting from this new estimate. By considering values *T*, *C*, *D*, and *F* for the "target" cell, the system establishes whether the required value for the "assumption" cell lies between *A* and *E* or between *E* and *B*. The process is then repeated with this new pair of estimates, and continues until the difference between the current two estimates is sufficiently small, compared to the estimates themselves. At this point the "assumption" cell is defined as holding this last estimate as a constant value, and the spreadsheet window is updated.

### 3.3. Cells containing structured objects

It was mentioned in connection with "which" queries that the value of a cell may be a tuple rather than a number or string constant. In fact, the value could also be a list. In micro-PROLOG the primitive relation

$X$ ISALL $(Y\!:\!Z)$

instantiates $X$ to the list of all terms $Y$ satisfying conditions $Z$. In accordance with the notation for non-numeric functions, a cell defined using the @ISALL function would contain the list of all solutions to a query.

Items like this might be much more extensive than could be fitted inside a cell, no matter what the width, and would often have to be truncated when displayed. To view the full contents, windows can be opened onto such cells by explicit command. Unless stipulated by the user, cell windows have a default position and size, and if more than one is open they may overlap.

Having introduced cell windows, a variety of other ways of defining cells emerge as feasible. Firstly, a cell can be defined by the "text about" function. If, for example, cell *G*6 is defined by

text about *F*4

then, provided a window has been opened for *G*6, the system will load a text file about the current value of *F*4 and will display it in the cell window. If there is much text, the user will be able to scroll through it in the window.

Secondly, if a cell has been defined by the "graph of" function, then a variety of different types of graph can be displayed in the corresponding cell window. One argument of this function is the type of chart required, and a second argument is the range of cells whose values are to be graphed. For example,

graph of (*B*4(*A*1.*A*10))

defines a graph of the range of values of *A*1–*A*10, whose type is given by the value of *B*4. Of course, if only one type of graph were required then the first argument could specify it explicitly, instead of by cell reference.

In some applications it may be desirable to represent the solution to a query in pictorial form. If a set of cells are declared to hold the components of a solution to a query, then a further cell can be defined as the "picture of" the solution comprising these components. At present, the system relies on being able to access a PROLOG "draw" program, specially written for each particular problem, which produces such pictures from the component values of solutions.

In a simple example of the design of two-roomed flats [5], a declaration was made that a particular set of cells were related in such a way that they held the components of a solution to the design problem—one cell holding the position of the front door, another holding the position of the back-room window, and so on. A further cell was defined as the "picture of" the solution, and a "draw" program was written for producing plans of flats from the component values. As each solution was found by the system its picture was displayed in a cell window.

For a graph or picture, different parts of the cell window display will correspond to the values of the different cells that had been referenced in the "graph of" or "picture of" definition. In many cases it is possible to move the cursor inside the cell window and to manipulate parts of the diagram, these changes being reflected back as modifications to the values of the corresponding referenced cells. Since the different referenced cells are themselves interrelated, such direct manipulations will often result in changes to related parts of the picture.

It is intended that LogiCalc will eventually incorporate the tools for producing "draw" programs interactively, and that the system will finally have the functionality, as a pictorial simulation tool, of Borning's "ThingLab" [6]. This will require the incorporation of powerful problem-solving methods, including constraint-satisfaction techniques.

Lastly, a cell can be defined to be a spreadsheet itself, viewable and manipulable only when the cell window is open. It is possible to specify that such a new spreadsheet will have certain properties, including sharing the same definitions for a particular set of cells as some other spreadsheet, or with the value of its cells being the sum of the values of the corresponding cells of two or more other spreadsheets. (An application of this feature is the consolidation of accounts.) It would be straightforward to ensure that particular spreadsheets were secure from view or access. In other respects, new spreadsheets defined in cells can be treated in the same way as the main spreadsheet. In particular, cells of a new spreadsheet can again be spreadsheets, giving LogiCalc a recursive structure.

## 4. POTENTIAL AREAS FOR DEVELOPMENT

The author is currently investigating two approaches for extending LogiCalc into more powerful decision support systems.

The first approach [7] recognizes that a limitation common to most financial planning systems, whether simple electronic spreadsheets or more sophisticated programs, is their reliance on the user of a two-dimensional tabular representation of models. A language is to be developed which would allow the financial planner to express his business situation in a more natural and unconstrained way. The user's input would be translated into a set of PROLOG clauses corresponding to his intended model, and the system would represent aspects of the model using a variety of modes including spreadsheet and graphics. The user could carry out the full range of modelling operations and instigate structural changes to the model, either through the medium of the modelling language or by direct manipulation of the spreadsheet or graphs.

The second approach, perhaps based upon more speculative ideas, is inspired by the concept of "User Assistant" or "Dialogue Manager" described by Bullinger and Faehnrich [8]. Amongst the various different modes of man–computer communication there is no best overall dialogue technique. Each mode has its own pros and cons with respect to particular types of user, task, and application. Furthermore, in some applications the ability to employ a mixture of modes can lead to a significant increase in expressive power. This point has already been recognized in the design of spreadsheets, where direct manipulation is combined with formal techniques to specify formulas. The task of a "User Assistant" is to manage different communication modes effectively, so as to produce such "symbiotic" dialogues We are investigating whether LogiCalc could be made more intelligent by being provided with a "User Assistant". This would take decisions to display graphs or natural language messages related to operations of the user, prompt the user in appropriate modes, and so on.

## 5. CONCLUDING REMARKS

Important benefits appear to have been gained by implementing this spreadsheet as a logic program. Firstly, the fundamental structure of the system has been laid bare, permitting clear and simple solutions to various design problems, and pointing to ways in which the functionality of the system can be extended through the generalization of existing concepts. Secondly, since LogiCalc is a rule-based system written in an AI language, the more difficult task of developing it into a truly "intelligent" decision support tool is made less onerous.

**Acknowledgements**

The author is grateful for several useful discussions with Marek Sergot and Phil Vasey of the Logic Programming Group at Imperial College, and is particularly indebted to the head of the group, Professor Robert Kowalski, for much encouragement and support. Thanks are also due to Brian Steel of Logic Programming Associates, who was responsible for the IBM-PC version of micro-PROLOG in which later prototypes of LogiCalc were implemented.

## REFERENCES

1. Kriwaczek, F. (1982) Some applications of PROLOG to Decision Support Systems. M.Sc. report. Imperial College of Science and Technology, London.
2. McCabe, F. G., Clark, K. L., and Steel, B. D. *Micro-PROLOG Programmer's Reference Manual.* Logic Programming Associates, London.
3. Horowitz, E. and Sahni, S. (1977) *Fundamentals of data structures.* Pitman, London.
4. Sergot, M. (1982) A query-the-user facility for logic programming. *Proc. European Conf. on Integrated Interactive Computing System,* (Stresa), Italy
5. Markusz, Z. (1977) How to design variants of flats using programming language PROLOG based on mathematical logic *Proc. Information Processing 77*. North Holland, Amsterdam.
6. Borning, A. (1979) ThingLab—a constraint-oriented simulation laboratory. Xerox PARC Research Report SSL-79-3, Xerox, Palo Alto.
7. Kriwaczek, F., Phillips, J. A., and Russel, M. J. (1985) An intelligent financial planning system. Proposal for research. Imperial College of Science and Technology, London.
8. Bullinger, H. -J. and Faehnrich, K.-P. (1984) Symbiotic man–computer interfaces and the user assistant concept. *Proc. 1st U.S.A.–Japan Conf. on Human Computer Interaction.* Honolulu.

10

# Representing Legislation as Logic Programs

M. Sergot
Department of Computing,
Imperial College of Science and Technology
London, UK

**Abstract**
The law is a rich and natural source of expert system applications of different and complementary types. In particular, legislation which is definitional in character can often be formalized as rules in logic programs so that, when executed by an augmented PROLOG system such as APES, it can be queried as though it were a data base. The system in turn queries the user for additional information which it needs, and it can explain and justify its conclusions in terms of the original legislation. The resulting system has many of the features associated with expert systems, but it can be regarded more usefully as a precise and executable specification of what the legislation tries to express. This suggests that executable formalizations can aid the drafting process itself, and that the techniques have application outside the law for formulating and applying regulations in all kinds of organizations.

## 1. INTRODUCTION

The law provides an ideal experimental domain for research in many areas of artificial intelligence (AI). It is a rich source of difficult and challenging problems which involve issues of knowledge representation, the analysis of natural language, and the automation of practical and common-sense reasoning. In contrast with many other experimental domains, however, the successful application of AI techniques in law will give rise to practical applications of immense social significance. Even now it is possible to build practical and useful systems requiring relatively unambitious techniques which are already well understood. This paper is concerned primarily with systems which represent the law as computer programs, as legal 'expert systems' which can apply the law to the solution of specific legal problems.

The rules and regulations which govern the running of all institutions and organizations have exactly the same character as legal provisions. The terms 'law' and 'legislation' are used here in this wider sense, to include the laws of organizations as well as the laws of a state. The domain of potential applications is consequently very wide. It suggests, in particular, a non-conventional approach to the construction of software for conventional data-processing applications. A payroll system could be based directly on tax and sick pay legislation, and could include, for example, a representation of the company pension scheme, the rules which govern holiday allocation, and promotion regulations. In the Logic Programming Group at Imperial College we have constructed a number of experimental systems treating fragments of these components, some of which are described in later sections of this paper.

The drafters of legislation, and of regulations more generally, are normally expected to formulate the law as clearly and precisely as possible. This makes legislation an ideal application for logic programming. Many regulations are 'definitional' in character and can often be formulated as rules in logic programs. This formalization can then be queried as though it were a data base and it in turn will query the user for information which it needs to solve a given problem. Before discussing this approach in more detail, however, it is instructive to consider first a different and complementary type of legal expert system.

## 2. LEGAL EXPERT SYSTEM 1

There is one sense in which the law as an application area is no different from any other domain which requires the application of specialized professional skill and expertise. One could imagine constructing an expert system which incorporates in rule-based form the expertise of an experienced lawyer, for the sake of example, that of an advocate who gives good advice to clients accused of criminal offences. Such a system might well include rules like:

```
x should plead guilty of y
   if x accused of y
   and x did y
   and penalty for y is light
   and not x has reasonable defence for y

x should plead not guilty of y
   if x accused of y
   and not x did y
   and penalty for y is heavy
   and x has reasonable defence for y
```

and many more. Auxiliary rules would be needed to define amongst other things what constitutes a reasonable defence and when and which penalties are associated with the various offences.

The rules above are expressed as logical implications of the form

*A* if *B* and *C* and . . .

These implications are Horn clauses, extended as in PROLOG to allow negated conditions in rules. This fragment of first-order predicate logic is termed here 'extended Horn clause logic'.

In the two rules above, the symbols 'guilty', 'not guilty', 'light' and 'heavy' are constant symbols, '*x*' and '*y*' are variables, and predicate symbols like

'. . . should plead . . . of . . .'
'. . . accused of . . .'
'. . . did . . .'
'. . . penalty for . . . is . . .'
'. . . has reasonable defence for . . .'

have been written in distributed infix form to aid readability.

Legal Expert System 1 is intended to represent an expert system in the 'classic' style of MYCIN [1] or PROSPECTOR [2]. It would be constructed by a process in which the expert's implicit, or even subconscious, problem-solving methods are made explicit and expressed in rule-based form. These rules then combine to give a system which simulates the problem-solving behaviour of the original expert.

It is important to note that the rules of Legal Expert System 1 express an expert's opinion of the law, and as such have no legal authority. The law itself, for example, may well not prescribe what constitutes a 'reasonable defence'. This is a concept which the lawyer finds useful when advising clients, and one which he has built up by experience over the years. Legal Expert System 1 incorporates little knowledge of what the law actually says: it is more concerned with simulating the legal problem-solving process than with reasoning explicitly about the law itself.

Legal Expert System 1 is imaginary, and it is worth digressing to remark on the practicality of the approach it represents. There is nothing to suggest that building expert systems for legal practice would be any harder (or easier for that matter) than the construction of similar systems which already exist in the more traditional domains like medicine or geology. Waterman and Peterson have described a system, LDS [3, 4], which they constructed in the programming language ROSIE for the problem of settling claims in product liability cases. LDS attempts to model how such claims are actually settled in practice. Although it is primarily a research tool for analysing the effects of alternative liability rules, Waterman and Peterson have indicated on occasion [3] that LDS could be adapted straightforwardly for use as a consultative expert system in its specialist domain.

This paper is not concerned with systems which simulate a lawyer's practical problem-solving skills, however, but with systems which aim to represent the law itself, for whatever purpose. Legal Expert System 1 has been presented at this stage to introduce a recurring theme: the idea of legal expert system, not as decision *taker,* but as decision-taking *aid.*

Legal Expert System 1, potentially at least, can give conflicting advice. In general, there will be rules which suggest that the client should plead guilty, and others which advise him to do the opposite. One could, as in many expert systems, associate some kind of certainty factor with each rule, and then provide in the system's inference engine some method of combining the various certainties to eliminate the apparent inconsistency, ranking the system's advice according to these certainty measures. In many applications this is precisely what is required: a system which will weigh up the conflicting rules and facts, arriving at a small number of specific recommendations with some indication of how good they are likely to be in practice.

This approach is not recommended for the imaginary Legal Expert System 1. Real experts, particularly lawyers and accountants do give conflicting advice, and for good reason. It is the client himself and not the expert who must make the final decision: in this example, whether to plead guilty or not guilty. The expert's role is to present the various alternatives, and to point out the consequences of deciding one way or the other. Ultimately, the client must decide.

Legal Expert System 1, it is argued, is useful precisely because it has the ability to generate conflicting advice. Of course, an expert lawyer whose advice is 'either plead guilty or plead not guilty' is of little use. But expert systems have an important and characteristic feature. They can explain their recommendations by showing the assumptions on which these recommendations are based. When the recommendations are in conflict, these explanations take on the character of arguments. As an aside, we should not be worried that a single set of logical sentences can produce conflicting conclusions. There is nothing logically inconsistent about advising someone that he should plead guilty for one reason, and not guilty for another.

Thus the output from Legal Expert System 1 is not simply 'guilty' or 'not guilty', but rather the whole set of arguments which can be constructed from the facts of a specific case, some arguing for and others against a particular decision. The user of the system, the client as it were, is free to examine these arguments, compare them one against the other, and choose to accept one, or reject them all in favour of a more persuasive argument from some other source.

It would be tempting, of course, to extend the system so that it could also assist in the evaluation of arguments. A real expert may want to indicate the likelihood that his assumptions, and his recommendations, are correct. It may be possible, for example, to estimate the probability that a particular defence will be accepted as 'reasonable' in court. One could envisage systems which apply (meta-level) criteria to distinguish weak arguments from strong, by considering, for example, the chain of reasoning involved, and the source of the various rules which are used. The evaluation of arguments is a separate exercise which needs further investigation, however, and it is not pursued further in this paper.

The importance of constructing arguments which are in conflict will be considered later when we look at the effects of vagueness and case law. For now, an example which contrasts directly with Legal Expert System 1 is given below. It is concerned with representing, not the legal problem-solving process, but the content of a piece of legislation.

## 3. LEGAL EXPERT SYSTEM 2

Legal Expert System 2 deals with entitlement of claimants to Supplementary Benefit, one of the Social Security benefits in the United Kingdom. The system was implemented by Peter Hammond [5] as a feasibility study for the United Kingdom's Department of Health and Social Security (DHSS), to demonstrate the usefulness of expert systems within the DHSS, and to estimate the effort involved in constructing such systems. The aim was not to build a system which would be used by DHSS officials in practice. Hammond was assisted by Ian Pickup, an expert on Supplementary Benefit from the DHSS, who supplied detailed knowledge of the relevant legislation and how it is interpreted and applied within the DHSS.

In Hammond's system, entitlement to Supplementary Benefit is described in extended Horn clauses, which are executed on a microcomputer by APES (Augmented PROLOG for Expert Systems) [6, 7]. APES is described below. The top-level rule for entitlement to Supplementary Benefit is expressed in the pseudo-natural language syntax accepted by APES, as

```
x is entitled to supplementary benefit
  if not x is disqualified by sex
  and not x is a juvenile
  and educational status of x is OK
  and x is a GB resident
  and x is excused or registered for work
  and x needs financial help
  and not x is disqualified by trade-dispute.
```

Here negative conditions like

'not *x* is disqualified by sex'

are treated by negation as failure [8]:

> not *P* holds if and only if all the ways of showing *P* fail.

Most of the conditions in the entitlement rule are themselves defined by lower-level rules. For example

```
x is disqualified by sex
   if sex of x is female
   and x has partner
   and x is living with partner

x needs financial help
   if capital of x is less than max-capital for supplementary benefit
   and requirements of x exceed resources of x
```

although some 'rules' have no conditions

```
max-capital for supplementary benefit is 2500
pensionable age for male is 65
pensionable age for female is 60.
```

Other conditions in the entitlement rule, such as

> '*x* is a GB resident',

and in the lower level rules,

> 'sex of *x* is *y*',

for example, require information about the individual claimant, and are not defined by lower-level rules.

Specific information about individual claimants can always be supplied as rules in a supplementary PROLOG data base. Usually, however, it is more convenient and more realistic if the system requests this specific information when it is required. This eliminates the need to supply all information in advance, and it frees the user from having to know precisely what information is relevant before the consultation begins.

APES [6, 7] is an extended PROLOG interpreter which incorporates Query-the-User [9], a model of input–output for logic programs in which the user is regarded as an additional source of information which can contribute to the problem-solving process. When APES cannot solve a problem itself for lack of information, it assumes the information is available from the user and it asks the user to supply it. In Query-the-User (and APES), the user is regarded as a kind of external data base.

APES also maintains a trace of its execution, providing explanations on request which can be examined interactively by the user. In logic-based systems like APES, these explanations correspond to logical proofs that the answers computed follow logically from the combined knowledge of the system and the information supplied by the user. APES itself is implemented in micro-PROLOG [10].

As with any logic program, an interaction with APES is initiated by a query from the user. In the following sample dialogue, user input is in boldface:

```
confirm (Peter is entitled to supplementary benefit)
    What is Peter's sex?  male
    How old is Peter (in years)?  23
    Is Peter in full-time education?  why
```

---

```
      if    Peter is not in full-time education
      then  educational status of Peter is OK

            not Peter is disqualified by sex
            not Peter is a juvenile, so
      if    educational status of Peter is OK
      and   Peter is a GB resident
      and   Peter is excused or registered for work
      and   Peter needs financial help
      and   not Peter is disqualified by trade-dispute
      then  Peter is entitled to supplementary benefit
```

---

```
    Is Peter in full-time education?  no
          .
          .
          .
    Is Peter currently involved in a trade-dispute?  no

Yes, I can confirm that
    Peter is entitled to supplementary benefit  how

  To deduce
    Peter is entitled to supplementary benefit
  I used the rule

    x is entitled to supplementary benefit
      if   not x is disqualified by sex
      and  not x is a juvenile
      and  educational status of x is OK
      and  x is a GB resident
      and  x is excused or registered for work
      and  x needs financial help
      and  not x is disqualified by trade-dispute

  I can show
    1. not Peter is disqualified by sex
    2. not Peter is a juvenile
    3. educational status of Peter is OK
    4. Peter is a GB resident
    5. Peter is excused or registered for work
    6. Peter needs financial help
    7. not Peter is disqualified by trade-dispute

Explain?  6
          .
          .
          .
          etc.
```

The clauses describing entitlement to Supplementary Benefit can function as a program to check an individual's claim. As Hammond points out, however, the same rules can be used for many other purposes: to determine who is entitled to Supplementary Benefit

find ($x$ : $x$ is entitled to supplementary benefit),

or to discover all the benefits to which Peter is entitled,

find ($x$ : Peter is entitled to $x$),

or to find juveniles who may be in need of financial help,

find ($x$ : $x$ is a juvenile and $x$ needs financial help).

Approximately 70 of Hammond's rules cover entitlement to Supplementary Benefit of about 90 per cent of all claimants. Deciding the claims of the other 10 per cent involves a great deal of detail, however, so that extending the system to cater for all claimants would increase the number of rules several times over. In fact, Hammond's 70 rules handle almost exactly those claims which are processed routinely, and ignore the problematic cases in which DHSS officials require assistance. A practical implementation for the DHSS (but not necessarily for anyone else) would have to be more concerned with the less routine cases. From the DHSS's point of view, however, the system did demonstrate what is possible in a limited amount of time.

The first version of the system, comprising about 50 rules (for 60 per cent of claimants), was produced in two days, including a half-day introduction to PROLOG for Ian Pickup. These rules ran in APES on a 64K micro-computer, although they were later moved to a 128K system. The number of rules was increased to 70 in two more days. In a subsequent exercise, the system was extended to calculate the amount of benefit payable and to determine the date on which the payments are made. The final system contained more than 200 rules and was produced in about one man-month overall.

Two points need to be stressed. Firstly, the hardware requirements for the system are extremely modest. The feasibility of building legal expert systems on small computers has since been demonstrated by a number of other pilot projects at Imperial College, some of which are referred to in later sections of this paper. Secondly, Hammond and Pickup were able to avoid many of the time-consuming problems associated with knowledge elicitation for expert systems, and this is reflected in the large number of rules they were able to formulate in the short time available to them. Their experience seems to confirm a commonly expressed opinion. The law, and regulation based organizations more generally, are a natural source of expert system applications. The attraction of the application domain is clear. By its very nature, the law is well documented, and the existence of regulations and reported examples makes the process of knowledge elicitation very much easier. Even if the documentation is not already in a form which can readily be expressed in computer-intelligible terms, it provides at the very least a convenient framework around which the knowledge acquisition process can proceed. This is not to say that there are no knowledge acquisition or knowledge representation problems.

Hammond's Supplementary Benefit system is an expert system in every sense. Knowledge is expressed explicitly in rule-based form; the system can request missing information which it needs, and it can explain and justify its conclusions. The knowledge which the rules express moreover is that of a human expert, i.e. Ian Pickup's opinion of what entitlement to Supplementary Benefit requires. This opinion is based on extracts from enabling legislation and various supplementary regulations, on a condensation of the relevant case law, and on familiarity with the DHSS's interpretation of the law and with its application in practice.

Consequently, the rules in Hammond's system are without legal authority. They express what Ian Pickup thinks entitlement to Supplementary Benefit requires, and not necessarily what the law actually says. So the question now arises: to what extent is it possible to eliminate the human expert from this process altogether and base a system directly on the actual legislation itself?

## 4. REPRESENTING LEGISLATION IN LOGIC

A part of the law is expressed as written provisions or explicit regulations, but the law in force at any one time is also determined by the decisions which were taken in previous cases. In many systems of law, case law has a legally binding nature through the doctrine of precedent. In other legal systems, and in organizations of all kinds outside the law, precedent may not be legally binding, but there is a role for case law nevertheless. The decisions taken in previous cases always have to be considered, if only for the sake of consistency and fairness. This paper concentrates on explicitly written legal provisions, although case law is mentioned also.

It is sometimes suggested that the law is also influenced by general legal principles ('no man may profit by his own wrong doing', for example). The legal status of such principles, and to what extent they must be considered in practice, remains a matter of debate for scholars of jurisprudence. In most cases, and certainly for the purposes of this paper, the practical effect of these vague principles can be ignored.

If a fragment of written legislation and its associated case law can be formalized in some mechanizable form of logic, then that formalization can function as a program which interprets and applies the law. A logical formulation of Social Security regulations, for example, can be regarded as a set of axioms. The entitlement of a claimant to a particular benefit is an example of a theorem we might try to prove from these axioms. As we shall see later, the proof of such a theorem (trivial by the standards of mathematics but useful nevertheless) is within the capabilities of even relatively unsophisticated theorem provers like PROLOG.

The advantages of this logic-based approach over the use of conventional programming languages are many. Legislation is not typically expressed as detailed algorithms: representing a complex and very high-level specification of some legal concept in a low-level algorithmic programming language is rarely a practical possibility. Symbolic logic, and at this stage we do not need to consider exactly which form of symbolic logic is appropriate, provides a precise language which resembles the draftsman's natural language closely. Symbolic logic, in the form of logical implications, can be regarded as the purest form of rule-based programming language, and the formalization inherits many of the advantages normally ascribed to expert systems. In particular, the use of a rule-based language makes the program comparatively easy to read, for lawyers and users alike. It is also easy to maintain and gives a natural way of generating explanations by constructing a trace of the rules which were used during execution.

Clearly these advantages are enhanced if the particular form of logic allows formalizations which resemble as closely as possible the style and structure of the original legislation. A close correspondence between legislation and its formalization increases confidence that the formalization is, in some sense, correct, and it makes the resulting program easier to maintain as the legislation changes.

It is possible in theory to build automated theorem provers for any system of symbolic logic; in practice, efficient proof procedures exist only for a small number of logical systems. Typically, theorem provers which are tolerably efficient are restricted to various fragments of first-order predicate logic. Extended Horn clause logic is one such fragment. It is the basis for the computational paradigm, logic programming, and for the programming language PROLOG. Indeed, PROLOG can be regarded as a simple theorem prover which incorporates various restrictions to make it an efficient executor of logic programs. PROLOG has a fixed method of executing programs and this imposes severe limitations on its use as a general-purpose theorem prover (it sometimes goes into loops, for example). Nevertheless, PROLOG can be used to derive many useful and non-trivial logical consequences from axioms expressed as extended Horn clauses.

Whether or not PROLOG will be adequate for the needs of a particular application will depend on the kind of problem-solving tasks we wish to support and on the style of the interaction we wish to provide. But there are reasons, independent of these considerations, to suggest that extended Horn clauses are an appropriate logic for representing many kinds of legislation. Extended Horn clause logic is a fragment of first-order logic which does not allow disjunctive conclusions in rules. But we can dispense with disjunctive conclusions for many practical purposes. In particular, rules with disjunctive conclusions

$A$ or $B$ if $C$

are seldom encountered in law. Legislation is normally concerned with specifying the conditions under which some definite conclusion can be made.

Dispensing with disjunctive conclusions in rules is an important simplification (it allows us to apply relatively efficient theorem provers), but we could not realistically expect to formalize any sizeable fragment of legislation in pure Horn clauses. At the very least we shall have to allow negated conditions in rules if our formalizations are to bear any resemblance to the original texts on which they are based. Allowing negated conditions in rules takes us beyond Horn clause logic, and would seem to re-introduce the need to reason with all of first-order logic. But there is a method of treating negation without going outside the capabilities of Horn clause theorem provers. Keith Clark [8] has shown that classical negation

not $Q$ is true if and only if $Q$ is false

can be replaced by negation as failure

conclude not $Q$ when all ways of proving $Q$ fail

whenever we can make a 'Closed World' assumption

anything which is unknown to be true is assumed to be false.

Negation as failure is the easiest type of negation to deal with computationally. It is also a type of negation which is often appropriate when reasoning with law. Given the regulations which describe entitlement to some benefit, it is natural to assume that the regulations cover all the possible ways of being entitled, that there is not some other way of becoming entitled which the legislators have not bothered to mention. This is precisely the assumption which justifies the use of negation as failure: if we cannot determine that a particular individual is entitled to the benefit by any of the routes explicitly mentioned, it is normal to conclude that the individual is not entitled to that benefit. There will be occasions, of course, when it is difficult or impossible to say that all the relevant legal provisions have been considered. In those circumstances the treatment of negation by failure is not justified, and we shall have to look to some other, necessarily less efficient, method for dealing with negation. The suggestion that we can often treat negation as failure is purely a practical one. Under very specific conditions, Clark's result allows us to read negation classically but compute with it by failure. Nothing in what follows turns on this, however.

I have suggested that disjunctive conclusions are seldom encountered in law, but I have to be more precise about what I mean in making such a claim. Suppose we have rules

*A* if *B* (1)

*A* if *C* (2)

which formalize some piece of legislation. Suppose we want to say further that these are the only ways of establishing conclusion '*A*' which the legislation allows. We can express this 'only if' information either in the object language by replacing the two rules (1) and (2) with the biconditional

*A* iff *B* or *C* (3)

or alternatively by stating in the meta-language that rules (1) and (2) are the only ways of establishing '*A*'. In both cases it is legitimate to conclude '*B* or *C*' given '*A*'. In the first case, '*B* or *C*' is logically implied by rule (3) and the assertion '*A*'. In the second case, '*B* or *C*' follows by meta-level reasoning: if '*A*' holds and rules (1) and (2) are the only ways that '*A*' could hold, then either '*B*' holds or '*C*' holds.

Clark [8] showed that for every proof of negation by failure (using 'only if' assumptions expressed in the meta-language) there exists a structurally similar proof of the same conclusion using ordinary negation (with 'only if' assumptions expressed in the object language together with appropriate axioms of equality and inequality). This is the 'Closed World Assumption' which justifies negation as failure.

In suggesting that negation as failure is often appropriate when reasoning with law, I am in effect claiming that the law often expresses 'only if' information, either explicitly or implicitly. If we choose to write 'only if' halves for rules (1) and (2) in the object language, we are adding a rule

*B* or *C* if *A*

to the formalization. This is a rule with a disjunctive conclusion. I am not claiming that such rules are not encountered in law nor that we shall never want to include them in our representations.

When we apply some piece of law, we have some facts about a particular case and we use the legislation to determine what legal consequences follow from these facts. A claim that disjunctive conclusions are seldom encountered in law means that the rules which are used for this purpose (whether we write them down or not) seldom have disjunctive conclusions. These are 'if rules', and there is little difficulty in identifying them in practice. When we construct a formalization of the law, we write down these 'if rules' (and it is suggested that extended Horn clause logic is a natural choice for the formal language in which to do so). In addition to 'if rules', we shall want to include 'only if' information, in general. If we are constructing a program which will only be used to apply the law, we can, if we choose, interpret negation as failure and thus in effect include the 'only if' rules implicitly. A suggestion that the law tends to be naturally expressed as 'if and only if' rules is an argument for the use of negation as failure, as long as we limit ourselves to programs which apply the law. If we want our representation of the law to be used for some other purpose (to assert, for example, that a particular individual is entitled to Supplementary Benefit and investigate what follows from this assertion), or if we choose not to treat negation as failure for whatever reason, then we shall want to express the 'only if' part of the law explicitly, normally in the object language. These 'only if' rules will almost always have disjunctive conclusions. If the formalization includes explicit 'only if' rules, then Horn clause theorem provers will not be adequate for our purposes. In what follows, I shall be less careful: I might claim that legal rules do not have disjunctive conclusions, or that extended Horn clauses can express many legal provisions; I shall be referring only to the 'if' part of the law.

That understood, I have suggested that many legal provisions will translate naturally into Horn clauses extended to allow negated conditions. There is a certain amount of evidence to support this claim in Ronald Stamper's work on the LEGOL project (see for example [11, 12]). One aim of Stamper's project was to design a computer language for representing legislation. Stamper developed a method of analysis based on various semantic considerations, and this semantic model in turn formed the basis for a computer language in which rules could be written to simulate the effects of legal provisions. The LEGOL language (in its executable versions) was based on relational algebra. It is therefore fairly straightforward to reinterpret LEGOL rules as Horn clauses (and it is argued elsewhere [13] that there are significant advantages in doing so). Although such a translation overlooks the emphasis placed by Stamper on LEGOL's semantic model, it does suggest an improved way of computing with LEGOL rules which is independent of what these rules might 'mean'. It can be no coincidence that LEGOL, a computer language specifically designed for representing legislation, can be reinterpreted as extended Horn clauses, and in particular that LEGOL made no provision for expressing disjunctive conclusions in its rules.

While there are many legal provisions which can be expressed in extended Horn clause logic, there are also many types of legislation which cannot. Even legislation which could be expressed in extended Horn clause logic is often written in a way that makes such a translation inconvenient or stilted. For example, many legal draftsman adopt a style in which a general rule is expressed in one place, and a number of additional conditions, particularly exceptions to it, are listed separately elsewhere. It is also common to encounter descriptions of rules in terms of other rules. An extract from section 16 of the United Kingdom's Income and Corporation Taxes Act 1970 states:

(2) Where the claimant under subsection (1) is a woman
- (a) the references in that subsection to the claimant's wife shall be construed as references to the claimant's husband, and
- (b) unless she is a married woman living with her husband, for the reference in that subsection to £320 there shall be substituted a reference to £355, and for references to £75 references to £110.

The subsection (1) referred to gives the conditions under which a person could claim a particular allowance against personal Income Tax. Clearly, subsection (2) describes a similar rule which holds only for women, not by stating its conditions explicitly, but by specifying a modification to an earlier rule instead.

These and other devices used by draftsmen indicate that we shall have to incorporate a richer variety of features into our representation language if we are to produce formalizations which keep recognizably close to the wording of the provisions they are supposed to represent. We can go a long way towards providing such a set of language features, however, by introducing a number of straightforward syntactic extensions to the basic extended Horn clause form. To take a simple example, extended Horn clauses do not allow disjunction in the conditions of rules. But a clause with a disjunctive condition

$A$ if $B$ and ($C$ or $D$)

is logically equivalent to the two clauses

$A$ if $B$ and $C$
$A$ if $B$ and $D$

This equivalence can be exploited by regarding the more concise clause with the disjunctive condition as syntactic 'sugar' for the two clauses without disjunction. This particular extension is straightforward to implement, and is a feature of most existing PROLOG systems.

There is, of course more to the development of a language for representing legislation than merely providing syntactic extensions to extended Horn clause logic. These syntactic extensions can only help in formalizing legislation which could be translated into extended Horn clause logic anyway, however inconveniently. But there are many kinds of legislation which could not be expressed directly in extended Horn clause logic. We should consider, amongst other things, how we might represent what is sometimes assumed to be the characteristic feature of law, the need to reason about concepts such as obligation ('Thou shalt honour thy father and thy mother') and prohibition ('Thou shalt not kill').

The law is sometimes regarded as a set of norms which state what must or must not be done under particular circumstances. This model of the law leads naturally to consideration of the 'deontic' concepts: obligation, prohibition, permission, and their various elaborations. Such considerations in turn have lead to occasional suggestions that some form of specialized 'deontic' logic is necessary to represent the law and to analyse the processes of legal reasoning.

It should be fairly clear that there are many areas of law where we could not hope to proceed without some way of representing the deontic concepts and computing with them. Specialized deontic logics attempt to provide a general theoretical framework for reasoning about these concepts, and a whole range of deontic logics of various kinds and levels of sophistication have been proposed and investigated. Most of these investigations, however, have not been concerned with computational issues. Efficient and mechanizable proof procedures for non-classical logics in general, and for deontic logics in particular, have still to be discovered. We are unable to choose a suitable deontic logic and expect to construct a computer-based reasoning system around it. An alternative (and arguably more realistic) approach is to attempt a formalization of the deontic concepts within a classical logic. This approach would sacrifice the conciseness of a specialized logic for the greater flexibility offered by describing the concepts explicitly, and it would allow us to exploit immediately the efficient proof procedures which are known for various fragments of classical logic. The treatment of deontic concepts within a computational framework remains an important topic for current investigations, and further discussion is outside the scope of this paper.

We have to recognize that some areas of law, for example those which demand a sophisticated treatment of the deontic concepts, would seem to be beyond the reach of current automated reasoning techniques for the present. But there are also many areas of law where existing techniques can be applied immediately, and which give rise to an enormous number of potential applications already. There are many legal provisions, particularly those of an 'administrative' nature, which can be regarded as little more than a precise definition of some legal relationship or property. To take a typical example, legislation which describes the circumstances under which an individual becomes a citizen of a country can be viewed as the legal definition of 'citizen'. If this legislation is complex enough, a system which formalizes the definition will have significant practical value. In the next section, a system which formalizes a definition of British citizenship derived from the provisions of the British Nationality Act 1981 is described.

We might argue that citizenship should be viewed as an obligation, on the Government and on those who fall under its authority, to recognize as a citizen any individual who satisfies the citizenship requirements, and to accord him all the rights and privileges associated with that status. This view of citizenship reintroduces obligations, and introduces concepts such as 'authority', 'right' and 'privilege'. If we want to reason at this level of detail there are fundamental philosophical issues to consider and, from a computational point of view, we have the attendant representational problems to contend with again. Nevertheless, the definition of citizenship, in itself, is unlikely to involve such sophisticated concepts. If the definition is complex enough, a system which restricts itself to representing the definition alone would still be of practical value, even though it does not begin to represent what the possession of citizenship actually means.

Deontic logics (whether special-purpose or not) attempt to provide a general theoretical basis for reasoning about the deontic concepts. For many practical purposes, however, it will be unnecessary to import the whole of this general mechanism into every given application. We might argue that entitlement to a Social Security benefit imposes in reality an obligation on the Government to make the appropriate payment whenever a valid claim is made. In practice, we can treat the conditions for entitlement as a definition. For we shall often want to discover whether an individual is entitled to benefit; we shall rarely want to proceed beyond that, to consider what this entitlement means, whether it really does impose an obligation on the Government, or what else we might be able to infer from such an obligation if it does exist.

It can be argued, therefore, that substantial amounts of legislation are, or can be taken to be, essentially definitional in nature. Much legislation can be viewed as a high-level specification of some legal relationship or property. This simplification does not eliminate all the difficulties, however.

An example of what can be encountered in legislation, suggested by

Trevor Bench-Capon, is provided by the regulations under which a woman could receive a 'Housewives Non-Contributory Invalidity Pension' (HNCIP), another of the United Kingdom's Social Security benefits (now defunct). A key provision in the regulations stated that a person would be entitled to HNCIP 'if she is incapable of performing her normal household duties to a substantial extent'. Like many legal provisions, the HNCIP regulations are at the same time definite, ambiguous and vague.

The regulations are definite because they state clearly what their conclusion should be: a woman who satisfies all their various conditions is entitled to the HNCIP benefit.

The regulations are ambiguous because there are two different ways of reading the condition 'incapable of performing... to a substantial extent'. The ambiguity is a syntactic one, caused by imprecision in the scope of the adverbial phrase 'to a substantial extent'. The easiest way of seeing the ambiguity is by bracketing the condition to make the scope of the phrase explicit. The condition could mean

'(incapable of performing...) to a substantial extent'

or it could mean

'incapable of (performing... to a substantial extent).

Problems arise because a person can be capable and incapable of performing a substantial amount of her normal household duties at the same time. A woman who is capable of performing only 40 per cent of her normal household duties is incapable of a substantial amount (60 per cent), and capable of a substantial amount (40 per cent). Under the first interpretation, she is entitled to the benefit. Under the second interpretation, she is excluded from the benefit because she can perform her normal household duties to a substantial extent, and she is therefore not incapable of doing so.

The ambiguity is a genuine one which came to light in the course of appeals against decisions. (Two different interpretations were adopted in two different parts of the United Kingdom. In Northern Ireland it was held that a woman who is capable of performing a substantial amount of her normal household duties is thereby excluded from the benefit. In the rest of the United Kingdom, it was enough to be incapable of a substantial amount for a woman to receive the benefit.)

The syntactic ambiguity in the HNCIP regulations is presumably an unintended one which was overlooked at the drafting stage. The example serves to illustrate an important mechanism of the law. One role of the courts, or whatever other arbitration procedures are established, is to remove accidental ambiguity when the circumstances of a particular case make it necessary (and only then). Eventually a problematic case whose outcome rests on the reading of the ambiguity comes before the appointed adjudicator, and a decision is taken one way or the other. This decision sets a precedent for the future and establishes one particular interpretation as the 'correct' one to take. This 'debugging' mechanism also operates in organizations outside the law, although it is often very much easier to amend the regulations themselves rather than to rely on case law to specify the accepted interpretation.

The HNCIP regulations are also typical of legislation in that they are vague. Nowhere in the regulations is it specified what the phrases 'substantial extent' or 'normal household duties' are supposed to mean. Unlike the unintended syntactic ambiguity, however, this vagueness is not accidental. The drafter of the regulations communicated precisely what he wanted to say by leaving some of the conditions vague. The general intention of the regulations is clear, although the specific meaning of the conditions will be dependent on individual circumstances. Normal household duties would be different for women who have children and for women who do not, for example: no legislator could hope to foresee every eventuality and make explicit provision for it. A legal draftsman is often reluctant to specify a piece of legislation in the finest detail, preferring that every individual case be judged on its own merits. Moreover, social attitudes shift, and superfluous detail in regulations can make the law unnecessarily inflexible and resistant to change.

In spite of the vagueness in the regulations, most cases of entitlement to HNCIP would be decided straightforwardly in practice. Assuming that the regulations had been suitably disambiguated, it would normally be clear whether a person should be regarded as incapable of performing her normal household duties to a substantial extent or not (at least, this is what the draftsman has assumed). There may be occasions, however, when some doubt remains. There may also be cases in which an appeal is made against a particular decision. In those circumstances, an adjudicating body at the appropriate level of authority will have to intervene. When such a case is brought before it, the adjudicating body will be forced to make a decision: whether, for example, a particular individual is or is not incapable of performing her normal household duties to a substantial extent. This decision sets a precedent. Similar cases in the future will have to be decided in a similar way, either because the precedent is a legally binding one, or because there is a social or moral obligation that persons in the same circumstances be treated in the same way. Through this mechanism, case law emerges which supplements the written regulations, and which gradually closes the definition of the legal concept which the draftsman had chosen to leave unspecified.

In practice, of course, no two cases brought before a court will be identical in every respect. The court may decide that a new case resembles some previous one so closely that the previous decision should apply in the new case also. Alternatively, the court may be persuaded that the new case, although similar in many respects to previous ones, is nevertheless different enough to require a different decision.

The decisions of a court hold only for those specific individual cases which have been tried. There is (usually) no provision to decide hypothetical cases. It follows that in any new case we cannot say with certainty what its outcome will be, although legislation and case law may provide good guidelines. Until the case has been tried, there is no fact of the matter. Legal concepts are thus often said to exhibit open texture: a legal concept is only precisely defined for those individual cases which have been decided; there is no precise definition for cases which have still to be tried.

Ambiguity, imprecision, and vagueness are always present in legislation and it is the resolution of problems which arise as a consequence that gives legal reasoning much of its own special character. In view of this it is sometimes supposed that symbolic logic has no place in legal reasoning because, it is suggested, logic is a language of such precision that it can only be applied to concepts which are sharply defined. Yet it is exactly the precision of logic which makes it an indispensable tool for analysing and reasoning with law. We cannot avoid ambiguity and vagueness in law, and we must not pretend that these problems do not exist when it comes to constructing computer representations of law. But we cannot even describe ambiguity and vagueness unless we have a language of sufficient precision. Similar arguments for the use of logic are often made by legal scholars. Layman Allen, for example, has advocated the use of symbolic logic for many years, as a practical tool for analysing and potentially simplifying the content of legal documents (see for example [14]).

The stylized form of natural language used by legal draftsmen is a notorious source of imprecision and unintended ambiguity. Recently there have been several attempts to prescribe language standards which will improve the precision of legal documents. Most notably, Layman Allen has proposed the use of a 'normalized' form in which to draft legislation [15]. Not surprisingly, this 'normalized' form is based on a disciplined use of logical connectives, with some suggestions for syntactic conventions to avoid the inadvertent ambiguities to which legal draftsmen are prone. None of this is to argue that draftsmen should be forced, or even encouraged, to eliminate all imprecision from legislation. Many regulations are effective precisely because they are vague and under-specified, and we have seen an example of this in the HNCIP regulations. Rather, it is unintended ambiguities which need to be eliminated from legal documents.

In effect, Allen's normalized form is an attempt to provide the draftsman with a semi-formal language with some degree of precision. Such recommendations are beginning to be adopted in the drafting of new legislation. As we begin to construct computer representations of the law, we must be aware of these recommendations also. We must eliminate unintended imprecision from our representations of the law, and we cannot possibly do so if the representation language which we use is already imprecise.

The key contention is that legislation can be represented in a mechanizable form of logic, and that extended Horn clauses are a natural form of logic to take for many kinds of simple legislation. In order to test this hypothesis, and to identify candidate features for any syntactically richer representation language, we formalized a sizeable fragment of real legislation, the British Nationality Act 1981. The formalization of the British Nationality Act also tests to what extent the simple execution strategy of PROLOG would be adequate to apply the law (a specific but typical use), and it serves to illustrate why the use of a formal logic is not incompatible with the open texture of legal concepts.

## 5. LEGAL EXPERT SYSTEM 3

Much of the British Nationality Act 1981 was formalized in extended Horn clauses by a student, Fariba Sadri, in the months of July and August 1983 with no expert legal assistance. A self-contained part of the Act runs in APES on a 128K micro-computer as a program which can apply the law in individual cases. The formalization of the Act is described in more detail in ref. [16].

The British Nationality Act was introduced to provide a new definition of British citizenship in terms of four new categories. The Act itself consists of five Parts and some supplementary Schedules. The first four Parts define the four categories of citizenship. The fifth Part and the Schedules include various definitions which are needed to understand the other four Parts. The first Part of the Act deals with British Citizenship proper: rules for its automatic acquisition (at birth, for example), conditions for entitlement to register or naturalize, and provisions for the renunciation and resumption of citizenship. Each of the next three Parts treats one of the other categories of citizenship (British Dependent Territories Citizenship, British Overseas Citizenship, British Subjects) in a similar way.

At the time it was introduced, the Act was a very controversial piece of legislation and we hoped that its formalization might clarify some of the issues which caused the controversy. More importantly for the experiment, we chose the British Nationality Act because it exhibits all of the characteristics of legislation, while at the same time allowing a number of considerable simplifications. I shall consider these various simplifications in the course of the next few sections. It is enough to remark here that the Act contains so much detail, and its various provisions interact to such an extent, that even experienced lawyers find it difficult to assess the implications of the Act for particular individuals. Nevertheless, most individual clauses in the Act are easy enough to understand in isolation: useful and non-trivial conclusions can be reached by a straightforward, mechanical application of the rules, so that a formalization of the Act does have some practical value.

The first clause of the Act deals with the acquisition of British Citizenship at birth:

> 1-(1) A person born in the United Kingdom after commencement shall be a British Citizen if at the time of birth his father or mother is
> (a) a British citizen; or
> (b) settled in the United Kingdom.

According to the Act, 'after commencement' means after or on the date on which the Act comes into force. This clause of the Act can obviously be formalized as Horn clauses, although it is less obvious what specific predicates should be chosen. Since elsewhere in the Act it is important to know the date on which an individual acquires citizenship, and the section of the Act by which he does so, we eventually decided on:

```
x acquires British citizenship on date y by section 1.1
  if   x was born in the UK
  and  x was born on date y
  and  y is after or on commencement
  and  z is a parent of x
  and  (z is a British citizen on date y by section y1
             or
        z is settled in the UK on date y).
```

Here I have used the syntactic extension referred to earlier, which allows disjunctions in the conditions of rules. Notice that the formalization makes the assumption, not stated explicitly in the Act, that a person who acquires citizenship by section 1-(1) does so at birth.

The possession of British Citizenship can be related to its acquisition by the rule:

```
x is a British citizen on date y by section z
  if   x is alive on date y
  and  x acquires British citizenship on date y1 by section z
  and  y is after or on y1
  and  not (x ceased to be a British citizen on date y2
             and y2 is between y1 and y).
```

A person can cease to be a British citizen by renouncing it or by being deprived of it. The condition

*x* is alive on date *y*

ensures that a person ceases to be a British citizen when he dies.

Notice, too, that the definition of British citizenship is recursive: whether one person becomes a British citizen by section 1-(1) can depend on the citizenship of another. Elsewhere in the Act, acquisition of citizenship by one section can be blocked by the acquisition of citizenship by some other section (although in general an individual can be a British citizen by more than one section of the Act). It has sometimes been suggested that a propositional logic is adequate for representing legislation. The recursive definition of British citizenship suggests that, in the case of the British Nationality Act at least, a propositional language is not adequate.

Section 2-(1) of the Act states that:

> 2-(1) A person born outside the United Kingdom after commencement shall be a British citizen if at the time of the birth his father or mother
>
> (a) is a British citizen otherwise than by descent; or . . .

Since section 2-(1) is labelled 'Acquisition by descent' in the original text, it was natural to assume that the section could be formalized thus:

```
x acquires British citizenship on date y by section 2.1
  if    x was born outside the UK
  and   x was born on date y
  and   y is after or on commencement
  and   z is a parent of x
  and   (z is a British citizen on date y by section y1
           and y1 is different from 2.1)
              or
           (. . .))
```

In fact, it turns out that 'British citizen by descent' does not mean British citizen by section 2-(1) ('Acquisition by descent'). 'British citizenship by descent' is defined explicitly, and differently, in section 14 of the Act. The rule for section 2-(1) is corrected by replacing the condition

```
(x is a British citizen on date y by section y1
 and y1 is different from 2.1)
```

by the condition

```
(z is a British citizen on date y by section y1
 and not z is a British citizen by descent on date y)
```

and formalizing

```
z is a British citizen by descent on date y
```

to reflect the definition in section 14.

We assumed at the beginning of the experiment, rather naïvely with hindsight, that it would be possible to formalize the Act by considering every individual clause in isolation. This would have meant that different persons could work on the various parts of the Act independently, requiring only that they agree a common vocabulary of predicate names. It turned out that this was impossible in practice. The formalization was forced to proceed by a process of trial and error. Sections encountered later in the Act would often indicate that our formalization of some earlier section was inaccurate or over-simplified. The modification of the rule for section 2-(1) in the light of section 14 is a simple example. Other examples are given in ref. [16]. The need for these revisions raises questions about what is the intelligible unit of law: is it a clause, a section, an Act, or 'the law' in its entirety? This is important if we are ever to say that a formalization represents a fragment of law, since there will be some minimum size of fragment which must be considered. I shall return to this question in later sections of the paper.

In spite of these various complications, sections 1-(1) and 2-(1) of the British Nationality Act translate fairly naturally into extended Horn clause form. Although not all of the Act is quite so straightforward (occasional logical complexities did have to be clarified, some of which are described in ref. [16]), the British Nationality Act is an unusually well written piece of legal prose with relatively few of the linguistic contortions which make so much legislation obscure. Most representation problems arose simply from the scale of the exercise. The Act comprises approximately 70 pages of densely written legal prose. This is big enough to cause severe difficulties in the formalization; it is fairly small in comparison with other areas of law, the Social Security laws of the United Kingdom for example.

The most obvious application of the formalization is to use it as a program which can apply the law in specific cases, to determine whether or not a given individual is a British citizen of some particular category, or to determine whether he is entitled to register or nationalize if it turns out he is not. The easiest way of implementing this is to execute the program in an augmented PROLOG system like APES, so that missing person-specific information is requested from the user, and so that explanatory proofs of conclusions are available if required.

Questions from the system appear in the stylized form which APES uses if no natural language alternative has been specified. I have omitted various complications to make the dialogue easier to follow, and I have included in the program the information that the Act came into force on 1 January 1983. In order to begin a consultation with the system we ask, for example, whether a particular individual is a British citizen by any section of the Act on a given date:

**confirm (Peter is a British citizen on date (16 Jan 1984) by section *z*)**
Which *X*: Peter was born on date *X*? **(3 May 1983)**

Is it true that Peter died before (16 Jan 1984)? **no**
Is it true that Peter was born in the UK? **yes**
Which $X$: $X$ is the father of Peter? **William**
Which $X$: William was born on date $X$? **(1 March 1952)**
Is it true that William died before (16 Jan 1984)? **no**
Is it true that William was born in the UK? **yes**
Is it true that William was found as a new-born infant abandoned in the UK? **no**
Is it true that William was adopted? **no**
Is it true that William was a citizen of the UK and Colonies on date (31 Dec 1982)? **yes**
Is it true that on date (31 Dec 1982), William had the right of abode in the UK under the Immigration Act 1971 as then in force? **yes**

.
.
.

Is it true that Peter has ever renounced British citizenship? **no**
Is it true that Peter has ever been deprived of British citizenship? **no**

Yes, I can confirm that
Peter is a British citizen on date (16 Jan 1984) by section 1.1 **how**

I deduced
Peter is a British citizen on date (16 Jan 1984) by section 1.1
from the rule

```
x is a British citizen on date y by section z
   if    x is alive on date y
   and   x acquires British citizenship on date y1 by section z
   and   y is after or on y1
   and   not (x ceased to be a British citizen on date y2
               and y2 is between y1 and y)
```

I can show
1. Peter is alive on date (16 Jan 1984)
2. Peter acquires British citizenship on date (3 May 1983) by section 1.1
3. (16 Jan 1984) is after or on (3 May 1983)
4. not (Peter ceased to be a British citizen on date $y2$ and $y2$ is between (3 May 1983) and (16 Jan 1984))

Explain? **2**

I deduced
Peter acquires British citizenship on date (3 May 1983) by section 1.1

from the rule

```
x acquires British citizenship on date y by section 1.1
  if    x was born in the UK
  and   x was born on date y
  and   y is after or on commencement
  and   z is a parent of x
  and   (z is a British citizen on date y by section y1
                 or
         z is a settled in the UK on date y)
```

I can show
1. Peter was born in the UK
2. Peter was born on date (3 May 1983)
3. (3 May 1983) is after or on commencement
4. William is a parent of Peter
5. William is a British citizen on date (3 May 1983) by section 11.1

Explain?

In examining a proof such as this, the user has a number of options at every stage. He can choose to examine the proof of any individual subcondition to any required depth. In the more recent implementations of APES, he can in addition request a justification for the particular rule used. This justification would normally take the form of supplementary documentation which explains where the rule comes from, how it relates to the original legislation, and what hidden assumptions have been made in its construction. A justification for the rule representing section 1-(1) of the Act, for example, would include the original text, and should point out the assumption, not explicitly stated in the Act, that a person who acquires British citizenship by section 1-(1) does so at birth. In this way, the user can examine the proof in as much or as little detail as he requires.

The user may also, if he wishes, ask whether an alternative proof exists for the same conclusion. It may be, for example, that Peter is also a British citizen by section 1-(1) because his father was settled in the United Kingdom at the time of his birth. He may be a British citizen by virtue of his mother's citizenship, or for any number of other reasons. Each of the alternative proofs can be examined systematically in turn.

If the citizenship of an individual can not be confirmed, the proof of the negative answer can be examined in similar fashion. Because APES treats negation as failure, proving a negative conclusion involves explaining why every possible way in which citizenship could have been established does not in fact succeed.

The output computed by the British Nationality Act program is not simply the answer 'yes' or 'no', therefore. More important is the set of proofs which can be constructed from the axioms in the formalization and the information supplied by the user. Normally in expert systems, explanations are included to increase the user's confidence in the system's conclusions. In legal expert systems, the ability to generate proofs is more fundamental: the construction of proofs is usually the principal aim of consulting the system. I shall return to the importance of proofs in the next section.

I should remark on the state of the implementation of the British Nationality Act program. At the time the formalization was completed only a small fragment of the Act could be loaded, together with APES, onto the small micro-computers on which micro-PROLOG was then available. In order to demonstrate the system we were forced to identify a self-contained portion of the Act. We chose those sections in Part 1 of the Act which deal with the acquisition of British citizenship proper, and the relevant definitions from Part 5 and the Schedules. We also had to add some general rules (for example, to define parents in terms of fathers and mothers), and some rules for handling calculations on dates. As of December 1983, this system (containing approximately 150 rules) ran in APES on a micro-computer with 128K bytes of memory. We have estimated that a micro-PROLOG system which could address 512K bytes would be sufficient to run the complete Act (containing about 500 rules). Such micro-PROLOG systems have since become available, although at the time of writing we have not transferred the whole formalization to these larger systems.

The secondary objective of the experiment was to test whether PROLOG's simple execution strategy would be adequate for applying the formalization as a program. Since we attempted to keep the formalization as close as possible to the original text, and we were not concerned at all with its efficiency, it is a little surprising that the formalization actually runs remarkably well as a PROLOG program. There are occasional inefficiencies, when PROLOG recomputes something which has already been established earlier in the computation. These inefficiencies could be eliminated straightforwardly by storing selected portions of the computation as 'lemmas'. There are also isolated parts of the Act which cause the program to loop when executed by PROLOG. Such loops, of which there are two in the fragment of the Act which we run, are not problematic and both can be eliminated by program transformation methods.

In the meantime, we have consolidated our experience with the British Nationality Act by applying the same techniques to a number of other examples in the legal domain. Some of these are mentioned in a later section. For now, it is more important to make a number of general remarks about the whole approach.

## 6. AXIOMATIC MODELS OF LAW

It might be tempting to assume that our formalization of the British Nationality Act is a precise representation of what the legislation tries to express. Arguably, it does represent exactly what the draftsman intended. Nevertheless, the law concerning British citizenship is not what the draftsman tried to express, but what the relevant authorities decide that he did express, and these two things might not coincide. To take a simple example, recall that our formalization assumes that a person who becomes a British citizen by section 1-(1) does so at birth. This is a very reasonable assumption, and it may well be what the draftsman intended. Nevertheless, it is not what he wrote, and it is always possible that a case will arise in the future to cast doubt on this interpretation. If such a case does arise, and if the court decides that an individual became a British citizen by section 1-(1) at some time other than at birth, then our formalization will be an incorrect representation of the law, whether it represents what the draftsman intended or not. It is a key rule in the interpretation of legal documents that the exact wording is critical. It follows that paraphrasing is forbidden. Yet this is precisely what we have done in formalizing the British Nationality Act: we have paraphrased the original text, albeit in a formal language.

The British Nationality Act program is not so very different from Legal Expert System 2 after all. Hammond's system describes Ian Pickup's opinion of what the law requires for entitlement to Supplementary Benefit. The British Nationalty Act program formalizes a particular interpretation, mostly Fariba Sadri's, of the definition of British citizenship as given in the British Nationality Act.

There is a difference between the two systems, however, and it is an important one. We might, for some given application, need to claim that our formalization of the British Nationality Act represents accurately the current state of the law. To substantiate this claim we would have to show the rules in the formalization to an expert lawyer, and even to the draftsman himself. Each individual rule would have to be examined and compared against the original text on which it is based, together with any additional assumptions which were made in its construction. It is because the rules in the formalization are explicit that such an assessment is possible at all; the assessment becomes a practical possibility because the rules are expressed in a form which resembles the structure of the original legislation.

It would be very much harder to assess the accuracy of the model in Legal Expert System 2. We can take it that the model is an accurate one because Ian Pickup is an expert on Supplementary Benefit. Nevertheless, although the rules in Legal Expert System 2 are also stored explicitly, they are far removed from the original sources on which they are based.

It would be difficult or impossible to estimate how well Ian Pickup's interpretation of the law corresponds to the actual legislation. The difference between the two representations is one of granularity. Pickup viewed the law as a whole; Sadri viewed it section by section. The difference is similar to that between a precis and a paraphrase: accuracy is always easier to determine in the latter case.

Given that some assessment of the rules is possible, an assessment of the system's conclusions follows immediately. Both the British Nationality Act program and Legal Expert System 2 are axiomatic systems which attempt to model some fragment of the law. In both cases, answers computed by the program are theorems, logical consequences of the rules in the formalization and the information supplied by the user. It does not follow of course that the conclusions produced by such a system are accurate. It does mean, however, that a conclusion of the system is guaranteed to be accurate, if the axioms in the system are accurate.

This is a fundamental point. It explains why such emphasis has been placed on the examination of proofs and on the importance of documenting the source of the rules in the formalization. Since proofs are guaranteed to be valid, the only remaining doubt concerns the accuracy of the rules and the accuracy of the information which the user has supplied. The proof serves to identify the assumptions on which the conclusion is based; the accuracy of these assumptions can then be examined in detail.

This is in contrast to a model of the law which is expressed in a conventional programming language or in a rule-based programming language with no formal properties. A conventional program does not represent explicitly the assumptions on which it is based. It cannot produce proofs of its conclusions for critical examination, and it is practically impossible to state how the answers it computes relate to the assumptions in its model. A rule-based program does make its assumptions explicit, but unless the computation proceeds by the application of sound rules of inference, there is no sense in which the accuracy of the conclusions can be said to follow from the accuracy of its rules.

We are attempting to construct computer programs which purport to represent legislation in computer intelligible form. These programs are executed mechanically, and it would be of little practical value if the answers they produce could not be said to mean anything precise. A claimant who is advised by computer that he is entitled to Supplementary Benefit must know what this advice is worth: whether it would stand scrutiny in a court of law, for example. There is a real danger that the advice of computer-based legal systems will be overestimated. We can guard against misunderstandings only by explaining to users what the computer-generated advice means in reality, and this explanation cannot be in terms of detailed computations which the program invokes.

It is sometimes assumed that the test of a legal expert system is how well it can predict judicial decisions in its own particular domain. To suggest this is to miss the point entirely. For it must be recognized that legislation is often imprecise, and that it is essentially open textured. The role of the judiciary is to resolve questions of law as they arise. And the resolution of these questions is fundamentally a matter of choice. A judge will decide one way or the other, not because there is a right answer, but because he is forced to make a decision, and because he will find the arguments presented for one decision more persuasive than the arguments for another. This does not mean that we can never predict judicial decisions. The majority of cases will be decided straightforwardly (for otherwise that fragment of law would be so intolerably inefficient that it could not survive long its present form). If a legal expert system could not anticipate decisions in the routine cases, we would be right to dismiss it. But decisions in cases which are not routine can not be predicted, because it is impossible to anticipate all the arguments which could be presented, and because we cannot be sure which arguments the judge will prefer (that is what stops a case being decided routinely). If we accept that the law is open textured, then there really is no right answer until the judge has pronounced his verdict.

This is not to say that a legal expert system which is capable of predicting judicial decisions would not be an extremely useful tool in legal practice, or even that it would be impossible to build. But such a system would have to manipulate statistics and likelihood estimates of various kinds, and it is not the type of system which is being considered in this paper. It would resemble rather the imaginary Legal Expert System 1 which was described in this paper by way of introduction. This paper is concentrating instead on how a set of rules (axioms) might be constructed to model some fragment of the law (or more precisely, given the essentially indeterminate nature of the law, some particular interpretation of the law).

The ability to generate proofs takes on additional practical significance if we consider the nature of the computer systems we are attempting to build. In the first instance we might want to construct a system which is intended to take legal decisions. Although we would be reluctant to have the British Nationality Act program decide whether or not individuals have British citizenship, and although we would not want Legal Expert System 2 to process routinely the claims which are made for Supplementary Benefit, it would be perfectly acceptable to have many para-legal decisions taken mechanically by machine in applications outside the law. A company might want to process by machine the expense claims of its employees, or their requests to take leave, or any number of other similar day-to-day decisions. Indeed, the vast majority of 'legal' decisions taken every day are of precisely this trivial (though not necessarily straightforward) kind. We are most of us willing to accept the legal decisions taken by the payroll system which calculates our salaries, and the tax and other deductions which it makes, even though it is normally impossible to question the system about the assumptions on which these calculations are based.

When decision-taking systems are able to generate proofs they are able to account for their decisions, and they become applicable in less trivial decision-making tasks. At the very least a proof will demonstrate that any decision reached was not an irrational one. A proof will also form the basis for an appeal should any particular decision be unpopular or considered unfair for some reason. Moreover, there are bound to be decisions which will be unacceptable for any number of reasons, whether we apply the law by computer or not. The examination of proofs for such decisions would identify areas where the existing legislation could be revised; it may even suggest ways of implementing these revisions so that unwanted decisions are less likely to occur in the future.

For most applications within the law, however, it will never be acceptable to have decisions taken routinely by machine, whether such a machine can explain its reasoning or not. Legal expert systems are more widely applicable if they can be used, not as legal decision takers, but as legal decision-taking aids. In this kind of system the construction of proofs is more than a convenient by-product: in many applications of legal systems it will be the proofs and not the conclusions which will be of primary interest. An individual claiming for some additional benefit will want to know what reasons he can adduce to show that he is entitled; an officer writing a submission to an appeal tribunal will want to know which rules he can cite to support the decision that he made; and an advocate who is preparing his client's case will want to anticipate the opposing arguments he is likely to encounter.

The proofs which are generated as a result of consulting the system will provide a framework for a legal argument, although this framework will have to be developed further before we have a convincing legal argument. A proof starts from some assumptions or premisses and moves by rules of inference to a conclusion. If we accept the assumptions then we cannot deny the conclusion. However, we can properly refuse to accept a proof by casting doubt on the assumptions on which it is based. The assumptions which appear in our system-generated proofs are either rules from the formalization itself, or information which is supplied by the user in response to questions from the system. Let us consider in turn how doubt might arise over these two types of assumptions.

In the first instance we might have included rules in the formalization which are inaccurate because we have misread or misunderstood completely the provisions of the legislation. More likely (and more difficult to detect) is the possibility that we have not taken into account the effect of (meta-level) rules of interpretation. Most legal systems prescribe rules which govern the interpretation of legal documents, and which constrain how legal documents are to be read. We could presumably guard against this type of inaccuracy by involving an expert lawyer in the formalization process, to advise on matters of interpretation and to identify the laws of interpretation as they apply. Even more problematic, especially if we adopt the computational expedient of treating negation as failure, is the very real possibility of overlooking some legal provision which is not mentioned explicitly in the fragment we are formalizing but which is considered as part of the same law nevertheless. (The British Nationality Act 1981, for example, amends various provisions in the Immigration Act 1971, although we would have no way of knowing this from reading the text of the Immigration Act alone.) The law concerning some particular matter is usually derived from a great variety of sources and we must be aware of the dangers of treating an individual fragment in isolation. Such considerations again raise the problem of identifying an intelligible unit of law which I alluded to when discussing our formalization of the British Nationality Act. Just as we could not treat an individual section of that Act independently of the rest, we could not safely represent any one Act, or any other fragment of the law, in isolation. Yet we cannot realistically expect to formalize the whole of 'the law'. We could only hope to eliminate these problems, or at least not to overlook them, by taking expert legal advice as the formalization proceeds.

A more interesting source of inaccuracy perhaps is the case of legislation which is ambiguous, particularly since this was identified earlier as a characteristic of all legislation.

## 7. AMBIGUITY

When we began the formalization of the British Nationality Act, we expected that the process of formalization itself would identify cases of imprecision and ambiguity which had been overlooked at the drafting stage. I have already mentioned an example of imprecision in the British Nationality Act: the Act does not state clearly the time at which an individual becomes a British citizen by section 1-(1) of the Act. If the formalization is to run as a program then we have to make some additional assumptions. We decided to assume that such an individual would acquire citizenship at birth. It could be argued, however, that citizenship imposes duties as well as granting rights; that duties cannot be imposed on a minor; and therefore that an individual acquires citizenship by section 1-(1), not at birth, but on the day he reaches full age. This might be a tenable argument (in fact, it is easily defeated by considering other sections of the Act), and, if it is, we have two distinct interpretations of the Act. If we cannot choose between them then we really have no choice: we must incorporate both interpretations of the Act together, or risk giving a distorted view of the law. We have two different, similar, but distinct, formalizations of the law. Now if we ask whether a given individual is a British citizen on a given date we may receive two different and conflicting answers: 'yes' by one interpretation (and an associated proof); 'no' by the other interpretation (and its associated proof). All else being equal, the proofs will differ where the rules for section 1-(1) of the Act appear. We are left with choosing whether we prefer one interpretation of the law to the other. If we genuinely cannot decide, we must wait for a suitable case to come before the appropriate adjudicating authorities, for they will be forced to make a decision which in turn will help us decide which of the conflicting interpretations to adopt. Of course, if an individual's citizenship is not determined by the application of section 1-(1), then the two formalizations will agree (although in general there will be other formalizations which disagree about the reading of some other section or some other ambiguous phrase in the Act).

Where there is genuine and obvious ambiguity in legislation, we have to formalize both possible interpretations or risk misrepresenting the law. Once a case has been tried and a suitable disambiguating precedent has been set, it may be possible to discard one or more of the formalizations. Until this happens we are left with several, potentially inconsistent, versions of the law. There remains the technical problem of maintaining large formalizations which are distinct but which contain an immense amount of duplication. I do not propose to discuss such (implementational) details in this paper.

It may happen of course that a piece of legislation is ambiguous, without this being at all obvious. The Housewive's Invalidity Pension (HNCIP) regulations which were mentioned earlier are an example. It is not clear on first reading that the regulations are ambiguous. Indeed, even though the ambiguity is known to exist (two different interpretations were adopted in practice), it is not easy to demonstrate where the ambiguity lies. The process of formalization does not in itself identify ambiguity or imprecision. The person who is responsible for constructing the formalization must detect the ambiguity before deciding how it should be treated. A representation language which is precise enough may allow him to describe as many different, unambiguous interpretations as he sees fit, but it does not in itself make the detection of the ambiguity automatic. It follows that we can expect cases to arise which will identify an ambiguity which was overlooked in the formalization. When such cases come to court, and as they are decided, we will be forced to amend our formalization of the law to reflect these decisions.

This raises a question concerning the formalization of regulations which quite obviously do not express what the draftsman intended. Bill Sharpe's account [17] of his attempt to formalize the United Kingdom's

Statutory Sick Pay legislation provides such an example. The Sick Pay regulations refer repeatedly to 'periods of incapacity for work' which are defined, in essence, as 'periods of four or more consecutive days, each of which is a day of incapacity for work'. Quoting now directly from Sharpe,

> we realize that . . . the rule does not say what it means. It is quite clear from the use made of the definition that a period of incapacity for work is not just a consecutive period of sickness but the longest such period. It begins when someone falls sick and ends when they are better; a subset of a period is not properly speaking a period in the sense meant there.

It may sometimes be impossible to devise a formalization which is at the same time an accurate model of what the draftsman intended and a faithful representation of what he actually wrote. I shall have a little more to say about Sharpe's example later in the paper.

## 8. VAGUENESS AND CASE LAW

Leaving aside inaccuracy in the formalization (although more could be said) let us consider now the accuracy of information supplied by the user. If the formalization is executed by a system such as APES, then, roughly speaking, the user will be required to answer questions about concepts which are mentioned in the legislation but which have not been defined explicitly by the draftsman. We can distinguish immediately two different types. There will be concepts which are undefined but which have a precise legal meaning nevertheless: although they are not defined in the fragment of legislation we have formalized, they are defined explicitly elsewhere. And there will be concepts which are not defined explicitly anywhere in the law, and which have no precise legal meaning.

Let us dismiss the less interesting type, the concepts which are defined precisely elsewhere. In the case of the British Nationality Act, there are many references to concepts from the Immigration Act 1971 and to concepts from previous Nationality Acts superseded by the 1981 Act. Presumably when the draftsman of the 1981 Act refers to 'a person who immediately before commencement was a citizen of the United Kingdom and Colonies', he means something very precise (although we would need to read the earlier Nationality Acts to find out exactly what it is). In formalizing the provisions of the 1981 Act there are several ways of proceeding. In addition to formalizing the 1981 Act, we can formalize the provisions of the earlier Acts too, to define precisely what it means to be a 'citizen of the United Kingdom and Colonies'. If this turns out to be impractical, we can adopt the technique of Legal Expert System 2 and build a definition of 'citizen of the United Kingdom and Colonies' from the opinion of some appropriate expert (although then we would get a system of less 'authority'). And if this is impractical too, we can always choose to leave the concept undefined, and treat it as if it were a concept of the second type, one with no precise meaning.

The more interesting concepts, perhaps, are those which are not defined explicitly anywhere in law. Into this category fall concepts which have a common-world meaning and which need no special legal definition (in the British Nationality Act, what it means to be 'born' for example), and vague concepts which have been left unspecified by the draftsman on purpose (like 'normal household duties' in the HNCIP regulations I mentioned earlier). The distinction is an artificial one of course, if we accept that the law is incurably open textured. With a little imagination, it is easy to see how a case could arise which would throw considerable doubt onto the meaning of 'born' (or, more likely, onto the meaning of phrases like 'the time of birth' or 'the place of birth'). Conversely, we shall sometimes be able to say with great confidence what 'normal household duties' are for a given individual. Nevertheless, we can take a pragmatic point of view, and distinguish informally what we might call 'concrete' data (like a person's date of birth) from concepts which are inherently vague (like 'normal household duties'). In treating these concepts we shall have to take into account the existence of any relevant case law, for this will constrain the meaning of the vague phrases. And we must always remember that a case in the future might force us to revise our classification of which concepts we consider 'concrete'.

So far we have been treating concepts which are not defined in the legislation by omitting them from the formalization altogether and by asking the user to supply information about them as it is required. But there is an alternative. There is a property which holds for any logical system (we might even take it to be the definition of a logical system) that

> when conclusion '*P*'
>   follows logically from assumptions $\{A1, \ldots, An, B\}$
> then conclusion '*P* if *B*'
>   follows logically from assumptions $\{A1, \ldots, An\}$.

If we are trying to derive conclusion '*P*' from assumptions $\{A1, \ldots, An\}$ (which are rules representing some piece of legislation) and cannot continue because some concept '*B*' appears in the conditions of a rule but is not defined, then we can add '*B*' to the assumptions $\{A1, \ldots, An\}$ and continue. If the computation now succeeds (that is to say if we manage to prove '*P*' from the augmented set of assumptions) then we will have proved '*P* if *B*' from the original assumptions $\{A1, \ldots, An\}$. In other words, we are able to report an answer which is qualified with this extra condition '*B*'. To take an example, instead of reporting 'Yes, Peter is a British Citizen', the system might respond with the qualified answer 'Yes, Peter is a British Citizen, if (it can be established that) he was a citizen of the United Kingdom and Colonies on 1 October 1982.' Let us call such a qualifying condition a 'tentative assumption'. It should be clear that we can qualify answers with as many tentative assumptions as we like.

Computation with tentative assumptions is not quite as straightforward as I have made out, particularly in the presence of negation as failure. There are some similarities to an extension to PROLOG proposed by Gabbay and Reyle [18] which they called 'hypothetical implication'. More detail is beyond the scope of this paper, but let us suppose that computation with tentative assumptions is available, because it gives us enormous flexibility.

In the first place we might try to implement a system by treating all undefined concepts as tentative assumptions. This does not give a system of much practical value unfortunately. For we would get from the system little more than a paraphrase of the original formalization, although without the original structure. (Consider what happens if we ask 'Is Peter a British citizen' from the formalization of the British Nationality Act. All we get in reply is the answer 'Yes', qualified with all the conditions which must be satisfied before Peter's citizenship can be established.)

Computation with tentative assumptions works very much better in conjunction with querying the user. In the case of the British Nationality Act, for example, it would be reasonable to ask users about the 'concrete' undefined concepts (a person's date and place of birth, the identity of his parents, and so on). The availability of this concrete information means that fewer sections of the Act will apply in a particular case. The system will produce answers which are more specific and which, if qualified, are qualified by low-level, detailed tentative assumptions.

Tentative assumptions can also be exploited to improve the treatment of queries to the user. We can allow users the option of refusing to answer a question which they do not understand or to which they do not know the answer, since the treatment of these 'don't know' answers clearly reduces to adding some appropriate tentative assumption. Once we allow users the possibility of answering questions with 'don't know', we can go further, and we can reasonably revert to the scheme in which users are asked to supply information about all undefined concepts, the vague as well as the concrete. For a user will sometimes be able to decide whether a vague concept applies in the particular case under consideration. If he cannot, there is always the option of answering 'don't know'.

I am not recommending any one of these possible treatments as the right one to adopt in practice. Which concepts to treat as tentative assumptions and which to treat by querying the user will depend on the particular application, and will probably reflect personal taste. The British Nationality Act program (or at least that fragment of it which we have been running on the small micro-computers) works reasonably enough by querying the user about all undefined concepts.

Whichever method we choose for treating undefined concepts, we must still help users to discover what these concepts mean. If the user is expected to answer questions in the course of the consultation then this help will have to be available when the question is asked. If we are using the mechanism of tentative assumptions then help can be delayed until after the consultation is complete. In either case, we have to give the user some indication of what the concept is taken to mean in practice, especially if there is existing case law which must be brought to his attention.

The help we provide can take a variety of forms, and the appropriate choice will depend on the nature of the application we are attempting to build. In the simplest cases, it might be enough to accompany every question to the user with a fragment of text which is intended to describe the concept in fairly general terms, and to summarize any relevant case law which might exist. Incidentally, this suggestion also allows me to make the point that it is not necessary to formalize every detail of written legislation. Section 50 of the British Nationality Act includes the information that 'ship includes a hovercraft'. It would be more sensible to include this as a footnote to questions about ships instead of going to the extreme lengths of including such information in the formalization. Similarly, one of the key conditions for acquiring British citizenship is that a person be 'settled in the United Kingdom'. Section 50 of the Act defines this concept as 'ordinarily resident in the United Kingdom without being subject under the immigration laws to any restriction on the period for which he may remain' (with some minor exceptions). Since 'ordinarily resident' is not defined further in the Act, there may be little point in formalizing the definition of 'settled in the United Kingdom'. Doing so would only replace questions about 'settled' with questions about 'ordinarily resident'. (Although it may be that 'ordinarily resident' is one of these concepts which are defined elsewhere in English law.)

Attaching explanatory text to questions will only provide adequate help in the simplest of applications. In general we shall have to give more detailed help. One possibility is to allow the user access to the extensive legal document retrieval systems which are now becoming increasingly important in everyday legal practice. We could also seek to advise on matters of case law by allowing the user to consult a data base of decisions which were taken in previous cases. We could construct a supplementary expert system (constructed by any suitable method) which would allow users to invoke a supplementary consultation whose sole aim is to help with determining what some vague concept has been held to mean in the past. Trevor Bench-Capon and I have proposed elsewhere [19] that vagueness in law could be captured by constructing a system of conflicting rules, arranged to generate arguments for and against a particular conclusion. These various methods of providing help are not mutually exclusive, of course.

In general, a system which represents some piece of written legislation will have a component which is a formalization of the written legislation, and a component which is designed to help with deciding the lowest level concepts whose definitions have been omitted from the formalization. It is critical that these components should be kept as distinct and separate as possible. For otherwise, by coupling the two components together, we get a system whose conclusions are based on one single set of assumptions. It becomes difficult then to disentangle those assumptions which are fairly certain (because they are derived from written legislation) from those whose accuracy is practically impossible to assess (since they express implicitly stated rules of law derived from fragments of case law). We must recognize that in questions about open-textured concepts there is no fact of the matter. Often, the best way of seeing why a concept is vague and discovering what kind of criteria are important for establishing its truth is by executing parts of the system several times over and varying the assumptions for each consultation. We can more readily encourage users to experiment in this way if we keep separate the different components in the system. There is a great deal of flexibility in all of this. We can allow supplementary experiments in the course of the main consultation. We can leave experimentation to the end of the main consultation, until we have an answer with its various qualifications. Or we can let the user decide which of these options he prefers.

A final remark is in order. If we are interested in building a system which is to take legal decisions autonomously (and I argued above that we might) then we *must* couple a formalization of written legislation with rules which define precisely how vague concepts are to be treated in practice. There are some applications where this could be done. But if we can write rules to eliminate vague concepts, then so could a draftsman. It is because we lose flexibility that autonomous and automatic decision takers are necessarily limited in scope.

It should be clear why our formalization of the British Nationality Act works reasonably well, both as a program for applying the law, and as a system for helping lawyers to solve legal problems related to British citizenship. The provisions of the Act are complex and interact to a great extent; the Act contains relatively few references to other legislation; and although vague phrases have been used by the draftsman, they are at the lowest level of detail and they only have to be taken into account once the 'concrete' items of data have been determined for a given individual. Not all legislation is like this however. If we take instead some fragment of legislation where the rules are very shallow, or where there are very few concepts which can be readily decided, then formalizing the legislation will give us very little help. We will be more exposed to the problems of vagueness and open texture, and we will have to rely to a greater extent on the supplementary advice giving systems which I sketched out above.

## 9. GOAL-DIRECTED FORMALIZATION

I have stressed several times that our formalization of the British Nationality Act should be regarded as an axiomatic system, and I have claimed that this is where all of its power is derived. However, the process by which the formalization was developed contrasts directly with the normal method of constructing an axiomatic system. Usually in mathematics an axiomatic system is constructed from the bottom up, by choosing first a set of primitive, undefined concepts, and then devising axioms which define new concepts in terms of those which are primitive or already defined.

This bottom-up approach is also found in attempts to represent legislation as computer programs, most notably in Ronald Stamper's work with the LEGOL project [11, 12], mentioned earlier. In LEGOL, representation of some piece of legislation proceeded in two distinct steps: first an analysis based on LEGOL's 'semantic model' attempted to identify the various entities, concepts and relationships which appear in the legislation; then LEGOL rules were written to simulate the effects of the legislation on the concepts identified in the analysis phase. Although a LEGOL program could not be described as an axiomatic system, the method of construction did correspond closely to the bottom-up construction of axiomatic theories in mathematics.

There are severe practical problems in attempting a bottom-up formalization of any realistically sized piece of legislation, however. For how are we to choose the bottom level primitives in the first place? In the case of the British Nationality Act, for example, we would be forced to decide at the outset whether a concept such as 'the Secretary of State's discretion', which occurs several times in the Act, is a primitive concept or one which should be defined in terms of something more primitive instead. And there are many other such phrases which appear in the Act.

I should stress that this is precisely the problem which the LEGOL project was attempting to address. Its primary objective was to develop a methodology for identifying the relevant concepts in a piece of legislation; the implementation of practical systems was a secondary objective. It might be argued that such an ambition is not a realistic one, but leaving that aside, it is certainly the case that no such complete methodology exists at present. We must consequently look to some other approach if we are to construct a formalization of some fragment of the law in practice.

The obvious alternative, and the way in which the formalization of the

British Nationality Act was developed, is to construct the formalization in a goal-directed, top-down manner instead. Thus, if our aim is to formalize how British citizenship is acquired, there is a natural place to begin. The conditions for acquiring British citizenship are given explicitly in the Act: formalizing them introduces lower level concepts which may or may not be defined elsewhere in the Act. We proceed accordingly, formalizing those concepts which are defined in the Act, and leaving out of the formalization those that are not (although more about this later). At every stage we have an executable version of the formalization, since top-level concepts are defined and systems like APES can query the user for information which is missing. More importantly, we only need to address the problem of representing a concept like 'the Secretary of State's discretion' when we know in what sense it is relevant to the top-most goals we are considering.

To see why this is important, note that the British Nationality Act does more than specify the circumstances under which an individual becomes a British citizen. From the Secretary of State's point of view, for example, the Act describes some circumstances under which he is permitted to treat one individual differently from another. If we are interested in representing this aspect of the Act also, the formalization will need to treat the Secretary of State's discretion in some detail, for the Secretary of State will be interested in establishing whether he is permitted to exercise his discretion in a particular case, and he will be interested in the conditions under which this permission is granted. For the purpose of establishing some given individual's British citizenship, however, this kind of detail is superfluous. If we want to establish an individual's citizenship, the Secretary of State's discretion can be treated as if it were a primitive concept (either he does exercise his discretion or he does not). The representation of discretion in all its generality is difficult. Top-down, goal-directed, formalization removes the need to address difficult problems of representation unnecessarily, until it becomes unavoidable to do so. This is a practical point. I am not suggesting that the representation of discretion, or the representation of any other general legal concept, is not a topic worth investigating in its own right.

Legislation is seldom written with a particular goal or method of application in mind, and we might argue therefore that its computer representation should not be tailored to a particular usage either. To some extent, a formalization in logic reflects this requirement. The British Nationality Act program is not limited to establishing the citizenship of a given individual. Any number of different queries will invoke a consultation of the system, and in this respect it resembles a deductive data base more than it does a traditional expert system (although if we are to use it as a deductive component to a data base we shall have PROLOG's limited execution strategy to contend with). And we could take the same formalization, make explicit the 'only if' rules we chose to treat implicitly in our program, and employ more powerful theorem provers to exploit the formalization for any number of different purposes.

Nevertheless, it is quite unrealistic to imagine that we can devise a formalization of some part of the law without taking any account of how we shall use the formalization once it is constructed. A single piece of legislation can be used for many different purposes: even to read its provisions we need some idea of what we want to know and the kinds of things we want to discover. In the same way, the formalization will be influenced by what we expect to do with the rules and the kinds of inferences we shall want to support, although we might have several goals in mind, and there might be several aspects of the legislation that we want to represent simultaneously. Written legislation is expressed in natural language which assumes (reasonably enough) an enormous amount of common-sense knowledge before any sense of it can be made at all. At the very least we shall need to ground the formalization in some minimal representation of the relevant common-sense knowledge. It seems inevitable that we shall have to make some assumption about intended usage in doing so. After all, we would not consider representing in computer-intelligible form some arbitrary piece of natural language text without having some idea of what this representation is for. Bill Sharpe makes a similar point, although in a slightly different way, in his account of the Statutory Sick Pay formalization [17] which I mentioned earlier. Sharpe found he could not represent the Sick Pay regulations without considering to some extent how his program would be used, and the kind of common-sense knowledge (he called it 'problem solving rules') which this would require.

In formalizing a piece of legislation, we shall sooner or later have to choose predicates, and we shall have to represent the legislation in terms of these predicates. In advocating top-down, goal-directed formalization, all I am suggesting is that we should not choose the bottom-level predicates at the outset, partly because we cannot know at that stage which common-world concepts will turn out to be relevant, and partly because we cannot realistically expect to represent the bottom-level concepts (common-world or not) without some idea of purpose.

The penalty for this goal-directed approach is the phenomenon of what Thorne McCarty has called 'the hyphenated predicate'. Thus in our formalization of the British Nationality Act there appear such monolithic 'primitive' predicates as 'had the right of abode in the United Kingdom under the Immigration Act 1971 as then in force', or 'was found abandoned in the United Kingdom as a new-born infant'. In his own work on the TAXMAN project (see for example ref. [20]), McCarty has stressed the need for some 'deep' underlying conceptual model without which, he argues, it is impossible to reason more than superficially about the law. This opinion is surely right, but it does not conflict at all with the goal-directed approach which I am advocating for the formalization of written legislation.

McCarty's TAXMAN work has been primarily concerned with representing and reasoning with case law. The TAXMAN experiments are based on cases covering various aspects of Corporate Tax Law in the United States, in which the central concepts are Corporations and Shareholders, stocks and shares, events and transactions, and, to some extent, permission and obligation too. A major concern of the TAXMAN work has therefore been the invention of an appropriate representation for these concepts, for without one it would be impossible to reason about these cases in any meaningful way. If we consider now the British Nationality Act, we see that high-level concepts like the acquisition of British citizenship are defined in terms of primitive concepts such as the time and place of a person's birth and the identity of his parents. It is these primitive concepts which form the underlying conceptual model for our formalization of the British Nationality Act (although it is such a trivial model that it hardly deserves the name). Between these two extremes, the TAXMAN programs with their rich conceptual model and the British Nationality Act program with its simple model, we might place Sharpe's formalization of Statutory Sick Pay legislation. The primitive concepts underlying the Sick Pay regulations are periods of sickness and incapacity for work. Although Sharpe did not have to represent the concept of 'sickness', he did have to consider how to represent periods of sickness. It seems almost superfluous to add that these three conceptual models, for the TAXMAN programs, for our British Nationality Act program, and for the Statutory Sick Pay program, have no primitive concepts in common.

How is it then that our formalization of the British Nationality Act contains primitive bottom-level predicates like 'was found abandoned in the United Kingdom as a new-born infant', for these are surely not part of our underlying conceptual model? We can distinguish, however, concepts which involve points of law from concepts which involve only points of fact. In our formalization, primitive 'hyphenated' predicates are used to represent facts which are certainly complex but which are devoid of any points of law. 'Hyphenated' predicates whose application depends on points of law are broken down further in terms of more primitive concepts. In a sense, the draftsman of the British Nationality Act forced us to introduce the predicate 'was found abandoned in the United Kingdom as a new-born infant' (or something like it) by introducing the notion in the wording of the Act. We are less likely to need such predicates when reasoning with case law because we would have no reason for introducing the corresponding concepts into our conceptual model.

Some 'hyphenated' predicates which involve only points of fact will sometimes have to be broken down to make their application more readily apparent. It must be recognized, however, that being found abandoned as a new-born infant is hardly a major consideration for establishing the British citizenship of a given individual. There are many such concepts in the British Nationality Act: they are not central to understanding the provisions of the Act, although they have to be considered eventually when we attempt to apply the Act in some specific instance. We do need a conceptual model if we are to represent some piece of law, that much is clear. But we only need a model which treats in detail the concepts which are central to understanding the particular fragment of law we are considering. It is unnecessary to insist that the model should be rich enough to treat all the peripheral concepts, too. We must be realistic, moreover. A deep, conceptual representation of what it means to be found abandoned as a new-born infant is staggeringly difficult, perhaps even impossible, and there are numerous similar concepts which appear in the British Nationality Act. It is impractical, and unnecessary, to insist on a conceptual model which is sophisticated enough to handle them all.

Of course there is more to constructing a conceptual model than simply compiling a list of suitable predicate names. We must capture also the various relationships and constraints which obtain between our primitive concepts (that fathers are always male, for example). If we are reasoning with representations of case law then it is these various constraints which will be of primary importance. In systems which represent written legislation, we shall be more concerned with rules which define the high-level concepts, but we cannot neglect the primitive concepts entirely. At the very least, the absence of a suitably rich conceptual model will lead to stilted dialogues; and there is always the danger of generating questions which will undermine the user's confidence in the system (when, for example, the system asks whether John, who is known to be male, is the mother of Mary). How to make use of constraints to eliminate such questions is outside the scope of this paper.

The concept of being found abandoned in the United Kingdom as a new-born infant can be treated superficially in our formalization of the British Nationality Act, not only because it is not central to understanding and reasoning about the conditions for acquiring British citizenship, but also because the concept has no special meaning in law other than the way we would naturally understand it as laymen (as far as I know). The application of the corresponding 'hyphenated' predicate does not turn on a point of law. Suppose however that a case should arise in the future concerning a child who was found in the street somewhere in the United Kingdom. Suppose moreover that there is considerable doubt whether this child should be regarded as a new-born infant, and whether or not it was actually found 'abandoned'. To decide the case, a court at some appropriate level of authority will eventually pronounce its verdict, and as soon as it does the whole situation will change. For it will no longer be the case that 'being found abandoned as a new-born infant' has only its common-sense meaning. The concept will have acquired a specific legal meaning too, constrained by the precedent which the court would have set. In these circumstances, we would be right to attempt a representation of what 'found abandoned as a new-born infant' means, and the underlying conceptual model would reflect the considerations which the court took into account when reaching its decision (the child's apparent age amongst other things perhaps). A similar situation, although a much less likely one, would arise if the Government ever decided to pass new legislation for the care of abandoned infants. A concept such as 'being found abandoned as a new-born infant', peripheral to understanding the British Nationality Act, would become central for the purposes of formalizing this piece of new legislation. But many of the problems with representing this concept would also have disappeared. In drafting the new provisions, the legislators would presumably have spelled out a precise legal meaning for this concept. This definition we could reasonably hope to formalize with some degree of confidence.

We must beware of assuming that a conceptual model which has allowed us to formalize one piece of legislation will allow us to formalize some other fragment also. Nevertheless, there may be some concepts which are fundamental to understanding the law and which are common to many types of legislation. Permission, obligation and their various elaborations are obvious candidates. Detailed investigation of such concepts and their representation in a computational framework are active areas of current research—for a recent (model-theoretic) treatment see McCarty's proposal [21].

In these last sections I have dwelt on some of the problems involved in representing legislation as computer programs. But we must not assume that these problems will always arise. There are many applications of legal systems where ambiguity, vagueness, and other types of indeterminacy in the law simply do not arise or where they can be discounted for all practical purposes. This is particularly true for applications in organizations outside the law. In the next section a brief account of a simple example of this kind is given.

## 10. LEGAL EXPERT SYSTEM 4

Following our formalization of the British Nationality Act, the same techniques have been applied at Imperial College to a number of other examples from the legal domain, mostly as student projects. Those which incorporate realistically sized fragments of legislation include a subset of the Immigration Act 1971 [22], regulations for Government grants to industry [23], and a large company's pension scheme with associated tax legislation [24].

The last example, dealing with pension regulations, is interesting because it incorporates an explicit representation of legislation within a system for decision support. It also illustrates the kind of application which can be implemented with relatively little effort.

The project was undertaken by David Chan as part of his M.Sc. course in the Department of Computing at Imperial College. The problem was supplied by a large chemical manufacturing company.

The company had recently introduced a scheme in which their employees could make an additional voluntary contribution (AVC) to the company's pension fund with a view to increasing their income after retirement. A problem arises because the Government in the United Kingdom imposes a limit on the total pension an individual can receive. This limit is determined for an individual by the salary received in the last few years before retirement, and by economic factors such as the rate of inflation prevalent at the time of retirement. Any additional voluntary contributions which raise an individual's pension above this limit are lost and cannot be refunded retrospectively.

Word had spread within the company that the AVC scheme was an attractive one, although in fact most of the company's employees were in such a position that AVC payments could not benefit them. Nevertheless, there were individuals who could benefit substantially from making some appropriate contributions under the scheme. The company was naturally concerned that those employees nearing retirement should receive the best possible advice concerning AVC. Local personnel officers seldom felt competent to give this kind of advice, however, so that the majority of queries were being referred to the AVC specialist in the company's Pensions Department. The classic need for an expert system arose as a consequence. AVC advice requires very specialized knowledge, yet the company AVC expert was so overwhelmed by requests that he was finding it increasingly difficult to cope.

At first Chan did consider implementing a classical expert system as a solution to the problem. He interviewed the specialist several times, and managed to formulate a number of suitable rules as a result. However, AVC advice is extremely difficult to give. It involves predicting, amongst other things, the likely date of retirement, the expected salary at retirement, the prospects for promotion or for exceptional bonuses in the meantime, and an estimate of what the rate of inflation will be like at the time of retirement, five or ten years in the future.

The expert was, naturally enough, very reluctant to propose possible rules. This reluctance was due partly to the difficulty of making the necessary predictions, but also because the expert was still himself in the process of building up expertise. Moreover, both the company and the expert were particularly anxious that no decision should ever be made on behalf of an employee. The decision, they insisted, must remain with the individual; the role of the expert was to explain the scheme as well as he could, and to help the client assess for himself the possible benefits of joining the scheme.

For these reasons, Chan rejected the classical expert system solution, and decided instead to base a system directly on a representation of the relevant regulations. This involved formalizing the company's internal pension regulations, the additional AVC regulations, and the relevant fragments of income tax legislation. Supplied with an individual employee's data, and an estimate for the rate of inflation and whatever other factors are relevant, the system calculates the total pension received under these assumptions. The employee is now encouraged to experiment with the various parameters. He can vary the date of retirement, his salary at that time, his monthly AVC contributions, the rate of inflation, and so on, until he acquires sufficient understanding of the scheme and how it relates to his own circumstances to make an informed decision. Explanations (proofs) generated by the system in support of its calculations also help in conveying to the employee what are the critical conditions for him.

Chan's system illustrates many of the points that have been made in this paper. The regulations on which the system is based are certainly complicated (the company has several separate pension schemes, for example), yet they introduce no particular logical complexities. The system incorporates one particular interpretation of the regulations: for all practical purposes there is little question that this interpretation is a reasonable one to take. The system itself is a combination of rules which formalize the written regulations, and supplementary rules which are derived from the opinion of an expert. There is no attempt to provide a system which will supply a particular recommendation. Instead, the user is expected to experiment with the system's various assumptions until he arrives at a conclusion of his own. And it may be that several mutually exclusive conclusions are equally likely. Deciding between them is then a matter of choice.

In this particular example the assumptions which the user is expected to vary are those relating to his own particular circumstances, and a value for the rate of inflation. In another example the state of the relevant legislation may be open to question instead. In principle, there is no additional technical difficulty in allowing users to vary also the rules which represent the legislation, to estimate the effects of taking different, mutually inconsistent, interpretations of the law. The law is ambiguous, imprecise and vague. But there is nothing logically inconsistent in contradictory conclusions which are derived from different assumptions.

And when these assumptions are equally plausible, deciding between them is at bottom a matter of choice.

## 11. CONCLUSION

In this paper I have discussed how it is possible to build an axiomatic model of some fragment of written legislation, and some of the ways we might use such a model. In particular, there are many areas of the law which we can reasonably regard as a set of definitions. Such legal definitions can be formalized fairly naturally as extended Horn clauses, although we will need to include a number of syntactic extensions to this language if the formalizations we produce are to resemble the style of the original legislation on which they are based.

How we proceed to apply such formalizations will depend on the nature of the application and on the way we plan to employ the system. In the first instance, we may be interested in constructing a system which is to take legal decisions autonomously in areas of the law which are not particularly sensitive. In most applications of legal systems, however, we shall be more concerned in providing legal decision support, either to help an adjudicator whose job it is to take decisions, or to aid an advocate in preparing and presenting his client's case in court. In all applications, I have argued, it is not so much the conclusions of the system which shall be of interest, but the proofs which such a system can construct.

Two contrasting and complementary methods for producing such a model of the law have been discussed. Hammond's description of entitlement to Supplementary Benefit was built by formalizing the opinion of an expert. Our formalization of the conditions under which an individual acquires British citizenship was based directly on the legislation itself. Recognizing that the law is often imprecise and ambiguous, and that it is necessarily vague, the difference between the two approaches is a methodological one rather than something more fundamental. There are practical differences between the two methods, however. We shall always have to justify our conclusions in terms of the assumptions on which these conclusions are based, and it is very much easier to assess the accuracy of assumptions when they can be traced to some authoritative legal source.

For the formalization of any reasonably large piece of legislation, for the purpose of building a practical application, I have advocated the use of a top-down, goal-directed approach. Any alternative would force us to identify the relevant bottom-level primitive concepts at the outset, before the formulation of rules can begin. Given the number of concepts which appear in any piece of legislation, common-sense as well as legal, I cannot believe that this would be a practical possibility in any but the simplest of applications. This is not to suggest that we should not look out for concepts which are common to many types of legislation, or to begin with concepts which we have formalized before and which we are confident of representing already. But there will always be concepts we have not encountered before, and which we shall not know how to treat. Goal-directed formalization allows us to delay addressing the most difficult representational problems until it becomes unavoidable to do so. And we always have the option of refining the formalization later by defining what was once a primitive concept in terms of something more primitive instead. We shall also have to ground our formalization by representing the common-sense knowledge required to understand and apply the legislation. It is very much easier to construct such representations when we know what kind of common-sense knowledge is required, and the ways in which this knowledge will be used.

Like all formalization, representing legislation in computer programs is necessarily a process of trial and error. This holds true whether the formalization proceeds from the bottom-up or whether we attempt to write rules from the outset. If we isolate an individual paragraph, or section, or page of legislation, trial and error creates no real difficulties. Difficulties arise as the scale of the representation increases, especially when we try to formalize a piece of legislation in its entirety. As the number of rules in the formalization increases, even the most trivial of adjustments becomes a major exercise, sometimes requiring large-scale revision and restructuring of the rules. If we are ever to make the formalization of legislation relatively routine we need techniques to manage these problems of scale. Modules and other methods of structuring provide only a partial solution. For it is often the case that the presence of a later section in some Act indicates that we have misunderstood or misrepresented an earlier section completely. I gave an example of this kind. Although section 2-(1) of the British Nationality Act is labelled 'Acquisition by descent', it is only when we encounter section 14 that we realize that 'citizen by descent' in earlier sections did not mean 'citizen by section 2-(1)' (as we might have assumed), but rather 'citizen by descent' as described in section 14 of the Act.

Since there is no alternative to trial and error formalization and since there seems to be no way of avoiding revisions as later sections are encountered, we need to provide a programming environment to make the incorporation of these revisions as painless as possible. I would suggest that in practice it is the absence of a suitable environment, rather than any shortcomings in representational languages or techniques, that imposes a limit on the kind of applications we can approach with reasonable confidence of success.

It is important to distinguish the use of extended Horn clause logic for representing legislation from the use of PROLOG to execute these representations. The example programs I have described in this paper are all executed by PROLOG (more precisely, by the augmented PROLOG system APES). In particular, these programs are executed top-down, reasoning backwards from conclusion to conditions. There will be many applications, however, where we shall want to execute programs with a mixture of backward and forward reasoning, or where we shall want to place the computation more in control of the user. One thing we shall never want to do is to execute these programs entirely bottom-up, supplying all relevant data in advance. This is a consequence of the very large number of detailed data items which are required for the solution of any given legal problem. To confirm my own British citizenship bottom-up, for example, I would have to anticipate the need for all kinds of data just in case they are required, ranging from the information that I was not found abandoned as a new-born infant, to the information that I have a reasonable command of English, that my father was never employed overseas in the service of the Crown, that I was never resident in the United Kingdom while in breach of the immigration laws as then in force, and many, many more such items of data. I might be willing to supply the date and place of my birth, the identity of my parents, and their date and place of birth, but I will not want supply much more than that in advance.

Vagueness and ambiguity are characteristic features of all legislation, but neither of them is an argument against the use of symbolic logic. Indeed I cannot see how we could begin to approach these problems without the precision of a formal language. It is no solution to invent a representational language which avoids these problems by ignoring them altogether. If a particular legal provision is genuinely ambiguous, and we manage to detect this ambiguity, we cannot ignore it. The law will have two distinct interpretations (or more) and, unless there are very exceptional circumstances, we shall have to make users aware of this fact and give them the opportunity of assessing the consequences of both interpretations. There may be reasons to prefer one interpretation over another, of course. We might prefer one particular interpretation because it is the more likely to be accepted in court; if we are given the job of preparing a client's case we might prefer one interpretation, not because it is the more likely to be accepted, but because it is in the interests of our client that our interpretation be accepted as the right one to take. How hard we shall have to work in persuading a court to accept our interpretation will depend the nature of the court itself, the ability of opponents in arguing the opposite, and on how many strong arguments we can adduce in support of our claim. In particular an interpretation which is far-fetched and which requires an elaborate and extremely involved argument to support it is unlikely to be accepted (although this does not mean it will not be accepted).

The resolution of vagueness too is fundamentally a matter of choice, constrained by the various rules of argumentation which a particular court will allow. In the long term, we shall need systems whose function is to advise on matters of interpretation, on the likelihood of getting some particular interpretation accepted, and on possible lines of argumentation to achieve this end. In the short term, I have mentioned a range of methods we could provide to help users arrive at some sort of decision. This help ranges from the simplest devices, like attaching some explanatory footnotes to questions about open-textured concepts, to sophisticated methods for accessing large data bases of decisions which were taken in previous similar cases. Trevor Bench-Capon and I are pursuing an approach which simulates the legal decision-taking process itself, by arranging for sets of inconsistent rules to generate arguments both for and against a particular conclusion. If all the generated arguments point in the same way then the decision will be relatively clear-cut. If not, it will be up to the individual user to assess the various arguments, and decide for himself which is the more likely to succeed in court, or which is in the best interests of his client. I have also suggested that we can go a long way towards conveying why a particular concept is vague by encouraging users to experiment with the various assumptions in the system. How much of this process we could mechanize, and what kind of advice we could provide about the best kind of experiments to perform, are the subjects of current investigation.

The use of a formal representation language gives the resulting program a meaning which is independent of its behaviour inside a machine. This property increases the usefulness of the formalization. Theorem provers no more sophisticated than PROLOG are adequate for executing the formalization as a program which applies the law. More sophisticated theorem provers can exploit the formalization for other purposes, particularly at the drafting stage and in the formulation of policy. The ability to prove logical consequences of a proposed piece of legislation can speed the drafting process and can improve the clarity of expression. The derivation of logical consequences helps to make clear whether proposed legislation is necessary or desirable. It can also suggest ways of simplifying or otherwise restructuring the legislation.

Let us imagine that at some point in the drafting of the British Nationality Act, concern was voiced that the proposed provisions were unfair because a person who was entitled to register as a citizen, but who died before he did register, could thereby deprive his children of British citizenship: but for his death, his children would be citizens. Suppose it is generally agreed that this state of affairs, if indeed it exists, is undesirable. Such concerns might be countered by suggesting that the current draft already has the property that the conditions under which a person is entitled to register as a citizen guarantee that his children are automatically citizens in their own right. If this is true, problems with persons who die before registering as citizens do not arise and no amendment of the current draft is necessary. Now this hypothesis might or might not be true. It could be tested straightforwardly however. Either it is a consequence of the current draft that entitlement of a parent to register as a British citizen implies the citizenship of his children, or it is not. Such a theorem, if it is a theorem, could be proved mechanically. If it is not a theorem, then the device which I called tentative assumptions would give the conditions under which it would be a theorem, and these qualifications in turn would suggest how the legislation could be amended to eliminate the anomaly of children being deprived of citizenship by the death of their parents. In principle there is nothing difficult about proving such theorems. In practice, the tools will only be used by a draftsman if they are convenient to use. This would be another motivation (if one were needed) for pursuing the development of logic programming environments.

As long ago as 1957, Layman Allen [14] advocated the use of symbolic logic as a tool for analysing, and potentially simplifying, the structure of complex legal documents. He suggested, in particular, the use of automated tools for this task, and automated theorem-proving techniques have advanced dramatically since that time. Mathematics is concerned with axiomatic systems in which a small number of very general axioms give rise to a large number of powerful and very general theorems. The kind of theorem proving which will be required for help in drafting the law will be in complete contrast: we shall want to supply a large number of very detailed axioms (representing some fragment of law), and derive very specific theorems from these axioms.

I have labelled the examples in this paper 'legal expert systems'. Whether such programs should properly be called expert systems is a matter of terminology. They certainly have many of the features associated with expert systems. They derive their conclusions by inference from knowledge expressed explicitly; they can justify their conclusions, and they can generate requests for missing information as it is required. But these programs also resemble the executable specifications common in software engineering, particularly if they are to be used at the drafting stage. This resemblance is more than superficial. I have mentioned that in formalizing the British Nationality Act we were not concerned with efficiency, but with keeping the formalization as close as possible to the structure of the original legislation. And I have also remarked that it may be impossible to achieve this, while at the same time producing a program which behaves well computationally. In the case of the British Nationality Act formalization, it is possible to employ program transformation methods to yield a program which is computationally better behaved, but which is less readable and further removed from the legislation as a consequence. What we shall want in such circumstances is a system which will execute the efficient but obscure program, but explain its conclusions in terms of the original, inefficient, formalization.

There are many areas in the law, and in regulation-based organizations more generally, for which it is possible to build applications in the short term. In this paper I have stressed the practical aspects, and I have mentioned only briefly the very many technical problems which remain to be solved. We need to discover natural and computationally tractable representations of the deontic concepts, for example, because they occur in many areas of law, although not nearly as frequently as might be supposed. Much more important, whether we treat the deontic concepts or whether we restrict attention to 'definitional' law, are the problems of legal reasoning which stem from conflicting beliefs and the need to reason with incomplete information. Experiments with varying assumptions are a crude attempt to deal with conflicting beliefs. Reasoning with incomplete information introduces default reasoning and reasoning with logical systems which are non-monotonic. The simplest way of providing a default, non-monotonic reasoning system is to adopt the treatment of negation as failure. This treatment of negation is only justified under very specific conditions however. In many practical applications these conditions will be satisfied. In general they will not, and we shall be unable to proceed without some more sophisticated treatment for making default inferences on the basis of incomplete information.

There is hardly an area of research in Artificial Intelligence which could not find in the law an immensely rich source of challenging problems. Unlike most other experimental domains, however, any advance finds immediate practical application, in a domain moreover where applications will have the profoundest social implications.

**Acknowledgements**

I am indebted to Robert Kowalski for his enthusiasm, encouragement and support, and for his many valuable suggestions. I am particularly grateful to Trevor Bench-Capon for his comments on this paper and for many stimulating and informative discussions, to Peter Hammond to whom much of the credit for APES belongs, and to Fariba Sadri for undertaking the tedious job of formalizing the British Nationality Act. I am also grateful to Thorne McCarty and Ronald Stamper for discussions which have helped me enormously. This work was supported by the Science and Engineering Research Council.

## REFERENCES

1. Shortliffe, E. H. (1978) *Computer-based medical consultations: MYCIN*. North-Holland/Elsevier.
2. Duda, R., Gashing, J., and Hart, P. (1979) Model design in the PROSPECTOR consultant system for mineral exploitation. In *Expert systems in the micro-electronic age* (ed. D. Michie). Edinburgh University Press, Edinburgh.

3. Waterman, D. A. and Peterson, M. A. (1980) Rule-based models of legal expertise. *Proc. First National Conf. on Artificial Intelligence*, pp 272–275. Stanford University.
4. Waterman, D. A. and Peterson, M. A. (1981) Models of legal decision making. *Technical Report R-2717-1CJ.* The Rand Corporation, Santa Monica.
5. Hammond, P. (1983) Representation of DHSS regulations as a logic program. *Proc. BCS Expert Systems '83,* Cambridge. British Computer Society.
6. Hammond, P. and Sergot, M. J. (1983) A PROLOG shell for logic based expert systems. *Proc. BCS Expert Systems '83,* Cambridge. British Computer Society.
7. Hammond, P. and Sergot, M. J. (1984) *APES 1.1 reference manual.* Logic Based Systems Ltd, Richmond, UK.
8. Clark, K. L. (1978) Negation as failure. *Logic and data bases* (eds H. Gallaire and J. Minker). Plenum Press, London.
9. Sergot, M. J. (1983) A Query-the-user facility for logic programming. In *Integrated interactive computer systems* (eds P. Degano and E. Sandwell). North-Holland, Amsterdam.
10. Clark, K. L. and McCabe, F. G. (1984) *micro-PROLOG programming in logic.* Prentice-Hall, London.
11. Stamper, R. K. (1979) LEGOL: Modelling legal rules by computer. *Computer science and law* (ed. B. Niblett). Cambridge University Press, Cambridge.
12. Jones, S., Mason, P. J., and Stamper, R. K. (1979) Legol-2.0: a relational specification language for complex rules. *Information Systems* **4**(4), 293–305.
13. Sergot, M. J. (1980) Programming law: LEGOL as a logic programming language. Department of Computing, Imperial College of Science and Technology, London.
14. Allen, L. E. (1957) Symbolic logic: a razor-edged tool for drafting and interpreting legal documents. *Yale Law Journal,* **66,** 833–79.
15. Allen, L. E. (1979) Language, law and logic. plain legal drafting for the electronic age. In *Computer science and law* (ed. B. Niblett). Cambridge University Press, Cambridge.
16. Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., and Cory, H. T. (1985) The British Nationality Act as a logic program. Research report, Department of Computing, Imperial College of Science and Technology, London (December 1983, revised May 1985). To appear in *Commun. ACM* **29**(5), 370–86.
17. Sharpe, W. P. (1984) Logic programming for the law. *Proc. BCS Expert Systems '84.* British Computer Society.
18. Gabbay, D. M. and Reyle, U. (1984) N-PROLOG: An extension of PROLOG with hypothetical implications. I. *J. Logic Programming* **4,** 319–55.
19. Bench-Capon, T. J. M. and Sergot, M. J. (1985) Towards a rule-based representation of open texture in law. Research report, Department of Computing, Imperial College of Science and Technology, London.
20. McCarty, L. T. (1979) The TAXMAN project: towards a cognitive theory of legal argument. In *Computer science and law* (ed. B. Niblett). Cambridge University Press, Cambridge.
21. McCarty, L. T. (1983) Permissions and obligations. *Proc. IJCAI-8,* Karlsruhe.
22. Suphamongkhon, K. (1984) Towards an expert system on immigration legislation. M.Sc. thesis, Department of Computing, Imperial College of Science and Technology, London.
23. Lowes, D. (1984) Assistance to industry: a logical approach. M.Sc. thesis, Department of Computing, Imperial College of Science and Technology, London.
24. Chan, D. (1984) A logic based legal expert system. M.Sc. thesis, Department of Computing, Imperial College of Science and Technology, London.
25. Sergot, M. J. (1982) Prospects for representing the law as logic programs. In *Logic programming* (eds R. Clark and S. Tarnlund). Academic Press, London.

# MACHINE LEARNING: METHODS AND INSTRUMENTS

11

# Incremental Learning of Concept Descriptions: A Method and Experimental Results

R. E. Reinke
GTE Laboratories,
Waltham, Massachusetts, USA
R. S. Michalski*
Massachusetts Institute of Technology,
Cambridge, Massachusetts, USA

**Abstract**

A system for learning concept descriptions incrementally is described and illustrated by a series of experiments in the domains of insect classification, chess endgames and plant disease diagnosis. The system employs a *full-memory* learning method that incrementally improves hypotheses, but does not forget facts. The method is used to form both *characteristic* descriptions, which describe a concept in detail, and *discriminant* descriptions, which specify only properties needed to distinguish a given concept from a given set of other concepts. Experimental results show the advantages of inducing and maintaining only characteristic descriptions during learning and creating discriminant descriptions from them when a classification decision is necessary.

## 1. INTRODUCTION

Research in the area of concept learning from examples has been concerned mainly with methods for single step, or non-incremental, learning. Such methods can effectively and efficiently induce good descriptions from a given set of examples and, optionally, induce counter-examples (for example Michalski, 1975, 1980a; Quinlan, 1979; Langley *et al.*, 1983). These methods cannot modify concept descriptions which are contradicted by new examples, but must re-learn the descriptions from scratch. In contrast, incremental learning methods modify concept descriptions to accommodate new learning events (Winston, 1975; Michalski and Larson, 1978).

When we observe human learning we clearly see that it is incremental. People learn concept descriptions from facts and incrementally refine

* Current address: c/o George Mason University, 4400 University Drive, Fairfax, VA 22030.

those descriptions when new facts or observations become available. Newly gained information is used to refine knowledge structures and models, and rarely causes a reformulation of all the knowledge a person has about the subject at hand.

There are two major reasons why humans must learn incrementally:

1. Sequential flow of information. A human typically receives information in steps and must learn to deal with a situation long before all the information about it is available. When new information does become available, there is rarely time to reformulate everything known about all the concepts involved.

2. Limited memory and processing power. People cannot store and have easy access to all the information they have been exposed to. They seem to store only the most prominent facts and generalizations, then modify the generalizations when new facts are available.

This paper describes a method for automated learning of concept descriptions from examples which is novel in its use of facts and of concept descriptions. We assume that in practical machine learning systems, only the first of the above constraints is important and that the second may be ignored. The fact that information arrives sequentially cannot be changed, as it reflects the nature of the world. On the other hand, storing and retrieving large amounts of information is not difficult for modern computers. We therefore investigate a *full-memory* incremental learning system which modifies concept descriptions to accommodate new information, but does not forget facts.

A concept description can be assigned a *type* based on two factors: purpose and form. A description's purpose is either to *characterize* or to *discriminate* (Michalski, 1983). A characteristic description of a concept is very specific and tries to capture all the known properties of the concept. Such a description is useful for building a detailed model of the concept and for teaching someone about the concept. On the other hand, discriminant descriptions are used to distinguish one concept from a given set of other concepts and contain only those properties of the concept which are necessary to make such distinctions. Characteristic descriptions attempt to distinguish a given concept not just from a known set of other concepts but from *any* other concepts. Thus, discriminant descriptions are dependent on the class of concepts under consideration while characteristic descriptions are not. In short, characteristic descriptions are used to describe and discriminant descriptions are used to discriminate. Section 2 gives more details and presents a classification of different types of descriptions.

The form of a concept description is directly dependent on the description language used. In the variable-valued logic used in this paper (see next section), a description may be either *conjunctive* or *disjunctive.* We therefore distinguish between four types of description: characteristic conjunctive (CC), characteristic disjunctive (CD), discriminant conjunctive (DC), and discriminant disjunctive (DD).

People are able to learn and use many different types of concept descriptions. Further, the type of description a human uses may depend on the situation. The learning method described here can also be used to form several description types; these may be used in different ways when learning incrementally. We describe experiments designed to test the effectiveness of the new learning method over different description types in different domains.

Section 2 describes the problem area and introduces the relevant terminology. Section 3 describes the new learning methods as they are currently implemented, and presents some possible extensions. Section 4 describes experiments designed to test the learning methods and Section 5 presents the results of these experiments. Finally, Section 6 discusses the implications of the results and some directions for future research.

## 2. TERMINOLOGY AND DEFINITIONS

This paper deals with that subset of learning from examples known as *symbolic concept acquisition. Givens* are observational statements which describe objects (situations, events, etc.) that have been preclassified by a teacher. From these, the learning system is to induce a *concept recognition rule.* If an object satisfies this rule then it is considered an instance of the corresponding concept (class).

### *Attribute*

Throughout this paper, we assume that all objects and concepts are described in terms of a finite number of discrete attributes (variables). Each attribute is assigned a finite *domain* from which it draws values and a *type* that characterizes the structure of the domain. In this study, we distinguish only between two types of attributes: nominal and linear. Nominal attributes have domains where there is no ordering on the values (e.g. 'colour') while linear attributes have domains in which the values are linearly ordered (e.g. 'length').

### *Event*

An *event* is a symbolic description of an object. In this work, an event is represented as a vector of attribute values and is associated with a single concept (class). We assume that each event specifies exactly one legal value for every attribute. If an event is used in the learning phase, the event is called a *training* (*learning*) *event.* If it is used for testing, then it is called a *testing event.*

*Selector*

A *selector* is a relational statement of the form [***x#R***] where ***x*** is an attribute, # is a relation (one of ≥, >, =, <, ≤) and ***R*** is a subset of the domain of ***x***. The selector [***x#R***] is said to be satisfied by the event ***e*** if the value of the attribute ***x*** in ***e*** has relation # with at least one of the values in ***R***.

*Complex*

A *complex* is the logical product (conjunction) of selectors. A complex is *satisfied* by an event if every selector in the complex is satisfied by the event.

*Concept description*

A *concept description* is assumed to be a *disjunction* of *complexes.* A description is *satisfied by* (covers) an event if at least one complex of the disjunction is satisfied.

*Decision rule*

A *decision rule* is an assertion of the form ***D*** ::> ***C***. Here ***D*** is a concept description and ***C*** is a class (concept) and ::> denotes the class assignment operator. ***D*** can therefore be viewed as an hypothesis describing ***C***. The rule above can be interpreted as 'If an event satisfied description *D*, then the event is an instance of concept *C*.'

*Star*

The *star* of an event ***e*** *against the set of events* ***F***, denoted ***G***(***e*** | ***F***), is the set of all maximal under inclusion complexes satisfied by the event ***e*** and not satisfied by any event in the set ***F***. Informally, a star is the set of all maximally general concepts which consistently characterize a given example.

*Completeness, consistency, and description types*

A concept description learned from examples is *complete* if it is satisfied by all learning events which are known instances of that concept. A description is *consistent* if it is not satisfied by any learning event which is an instance of any other concept. Michalski (1983) defined a characteristic description as an expression that satisfied the completeness condition or the logical product of such expressions while a discriminant description is an expression that satisfies the completeness and consistency conditions or the logical disjunction of such expressions. Ideally, a learning system would learn the *maximal* characteristic description and the *minimal* discriminant description. In this section we will make a further distinction between conjunctive and disjunctive characteristic descriptions. A characteristic concept description is either a single conjunct listing the common properties of all learning instances of that concept or a disjunction of conjuncts which splits the learning instances of the concepts into subclasses. A characteristic disjunctive (CD) description should contain the minimum number of disjuncts and each disjunct should be as specialized (i.e. long) as possible. Note that the disjuncts in a CD description may not be disjoint and that the completeness condition still must hold.

## 3. METHODS AND IMPLEMENTATION

This section describes in detail the methods developed to learn descriptions incrementally from examples. Section 3.1 presents a very brief sketch of the AQ algorithm (see Michalski, 1975), as it is the base on which the method is built. Section 3.2 describes the modifications necessary to make AQ work incrementally with full memory and introduces an implementation of this method in the GEM program. Finally, Section 3.3 discusses a way to make GEM produce characteristic type descriptions.

### 3.1. The AQ algorithm

The AQ algorithm was conceived as a quasi-minimal solution to the general covering problem (Michalski, 1969). It has subsequently been recognized as applicable to the problem of inductive inference. This problem can be characterized as follows:

*Given*: a set of positive events $E^+$ belonging to the class for which a description is to be formed, and a set of negative events $E^-$ belonging to other classes.

*Produce*: a description $H$ that is satisfied by (covers) all the events in $E^+$ and none of the events in $E^-$.

A simplified version of the AQ algorithm applied to this problem randomly selects a seed event from a given class and generates the star for this seed. During star generation, the seed is generalized against different negative events. The results of these generalizations are intersected together to form a partial star. For efficiency reasons (Hong and Michalski, 1985), the partial star is reduced by selecting from it the preferred complexes as determined by a user-generated preference criterion. Once the reduced star is completed, the best complex in it is selected using the same criterion. The positive events covered by this complex are removed from the list of events to be covered, a new seed is selected from the remaining positive events and the process repeated. Stars are generated until there are no positive events left to cover; the disjunction of the generated complexes is a solution to the problem.

The preference criterion mentioned above is called the LEF (*lexicographical evaluation functional*). A LEF consists of an ordered set of criterion-tolerance pairs. A criterion specifies a metric to be used in judging complexes and a tolerance specifies the estimated relative error in that metric. When selecting the best complex from a list of complexes, AQ orders the complexes based on the first criterion. Complexes that are within the first tolerance of the best complex are ordered by the second criterion, and so on. The LEF provides a means of manipulating the types of descriptions produced by AQ (see Section 3.3).

### 3.2. Incremental learning with AQ

This section discusses extensions to the AQ algorithm which permit it to form descriptions incrementally (Becker, 1985). As shown in Figure 1, the modified algorithm must be able to apply inference rules either to training examples alone or to training examples and rules. Figure 2 shows a schematic version of the rule-modification process. The incremental method must be able to both specialize a rule so that it no longer covers a negative event and generalize a rule so that it covers a new positive event.

The incremental version of AQ begins by checking each old rule against the new events. It first determines whether any complex in these rules must be specialized. If some complex covers events which it should not, a modified version of AQ is invoked. The modified AQ procedure is

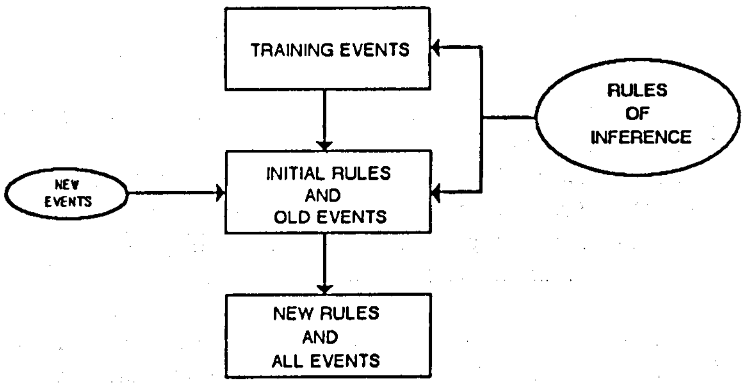


Figure 1. The initial steps in an incremental learning process.

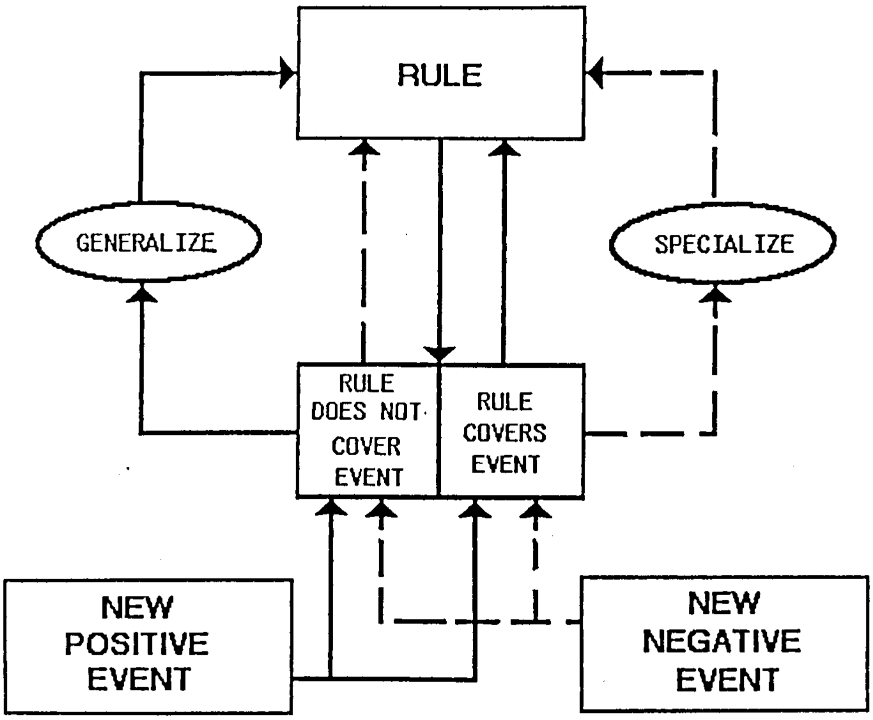


Figure 2. A schematic view of rule modification.

characterized below:

*Given*: a set of positive events $E^+$, a set of negative events $E^-$ and a subset of the event space, *SES*.

*Produce*: A description *H*, logically contained in *SES*, such that *H* covers all the events in $E^+$ and none of the events in $E^-$.

This is accomplished using the normal star generation technique, except that the first partial star is intersected with *SES*.

So, to specialize a complex, incremental AQ calls the modified algorithm with the following arguments:

| | |
|---|---|
| $E^+$ | all positive events (both old and new) covered by the old complex |
| $E^-$ | the new negative events covered by the old complex. |
| *SES* | the old complex. |

The result is one or more new complexes, all contained in the original complex, which cover all the positive events that the original did and none of the new negative events. This is the desired specialization.

Once all rules have been specialized, they are re-generalized to cover new positive events. This is done using the standard AQ method, except that the original rules are used as seeds. The result of this second step is a rule which correctly covers both old and new events.

The potential danger here is that the time spent finding every positive event covered by a complex during the specialization step will negate any time gain caused by the retention of old rules. Further, it is possible that the specialization process will produce unduly complex rules by splitting conjuncts into disjuncts. The experiments described in Section 4 were designed to address these issues.

The incremental version of AQ has been implemented in Pascal for efficiency reasons. The program, called GEM (Generalization of Examples by Machine) consists of approximately 3500 lines of code. All input to GEM is in the form of relational tables, allowing the program to interact with the QUIN relational data base system (Spackman, 1983).

## 3.3. Producing characteristic descriptions with GEM

The LEF (Section 3.1) used by GEM to choose the best complex in a star can be used to manipulate the type of description learned. Typically, the first criterion in the LEF is based on the number of positive events covered by the complex. The second criterion (used to break ties in the first) may be based on the length of the complexes. If the criterion requires that the best complex is the shortest, then GEM will produce discriminant descriptions. If the criterion requires that the best complex is the longest, the result is a more detailed, characteristic type of description. Since the program must sometimes create disjunctions in order to cover all positive events, the result of learning is a CD or DD description (although conjunctive descriptions can result). Two issues must be addressed: how good are these descriptions and what is the best way to use each type in learning?

The quality of a concept description depends on its performance and its comprehensibility. Both characteristic and discriminant descriptions should perform well when tested on previously unobserved events. A good discriminant description will also be easy to use (i.e. brief) while a good characteristic description will be detailed yet easy to understand. The comprehensibility of a description is obviously a subjective matter, but it is very important. If, for example, the descriptions are to be used in an expert system, the domain expert must be able to understand the results of learning.

There are many ways to use different concept description types in learning. The most obvious way is to simply form the type of description desired at whatever time it is needed. Another possibility is to learn incrementally only characteristic descriptions. This method is attractive for two reasons. First, characteristic descriptions are more specific than discriminant descriptions; a specific description contains more information about what is being learned and is less likely to be over-generalized. Second, since GEM can induce over descriptions as well as over events, it may be possible to induce good discriminant descriptions from characteristic descriptions. This second induction step should be very fast, and will allow us to use whichever description type is most appropriate. The question remains as to the quality of discriminant descriptions produced in this way.

## 4. EXPERIMENTS

In order to test the new incremental learning methodology, three application domains with differing properties were chosen. These domains (described in Section 4.1) varied in size, in type of attributes, and in the degree to which events represented real world objects or situations. This range of problems provides a basis for our tentative conclusions about the effectiveness of the learning methodology. An experiment, to be repeated in all three problem areas, was designed with the following goals in mind:

1. To compare the usefulness of different description types produced by the new incremental learning method.
2. To discover whether the method of inducing discriminant descriptions from characteristic ones produces simple discriminant descriptions that will perform well.
3. To see whether the incremental learning algorithm described in section 3.2 avoids the potential problems in learning with full memory.

### 4.1. Problems

The first problem was the classification of different species of *Stenonema* mayfly nymphs (Lewis, 1974) based on the use of attributes for describing an individual insect's appearance. Seven species of Interpunctatum group nymphs were described in terms of seven attributes, giving a total event space size on the order of $10^6$ possible descriptions. Ten different examples of each species were available.

The second application area was the King–Pawn–King black-to-move chess endgame, where the pawn's side is white. Here, examples were described in terms of 31 boolean attributes (Niblett, 1982); each example actually covered several legal KPK positions. That is, the input examples are generalized representations of the actual board positions. The examples were correctly classified (by a search program) into *Won* for the pawn's side or *Drawn*. A total of 1901 attribute vectors sufficed to represent the entire event space (which has on the order of $10^5$ positions) since one attribute vector represents many positions and a large portion of the attribute space consists of illegal, impossible, or symmetrical positions.

The largest application area was the soybean disease diagnosis domain. Diseased soybean crops were described in terms of 50 attributes.

Attribute domains ranged in size from two to 11 values, meaning that approximately $10^{30}$ attribute vectors were possible. The event set consisted of examples of 17 different soybean diseases common in Illinois; there were 17 different examples of each disease. The data used for these experiments differed from that described by Michalski and Chilausky (1980). For the current experiments, 15 more attributes were used and two new diseases were added to the data. The entire example set was also revised and updated.

### 4.2. Experimental method

To determine the quality and the usefulness of the full memory incremental learning method, an experiment was devised to simulate rule base development. In each problem area, all the available events were split randomly into two groups, training events and testing events. The basic learning method was to provide GEM with successive sets of new training events, so as to simulate rule base refinement. At each step of the process, the induced rules were tested on available testing events.

In each domain, the incremental learning process started with about 20% of the available learning events. Using this learning set, decisions rules were formed. An enhanced set of learning events was created by adding a random number of learning events of each class to the original learning set. The enhanced event and the rules induced during the first step were input to GEM, which then produced refined rules. The learning set was again enhanced, and new rules produced. This process was repeated until no learning events remained.

In the mayfly nymph domain, for example, there are seven classes and a total of five available learning events per class. The initial learning set was seven events, one per class. From these seven events, rules were induced. Then, seven random numbers were generated, one for each class. The results of this process are shown in Figure 3. For class *Stenonema carolina* the random number was 0.32. There were four events remaining in this class, so one example ($4 \times 0.32 = 1.28$, rounded to 1.0) of a *Stenonema carolina* mayfly nymph was added to the learning set. For this second learning step, a total of seven events were added. So, 14 events were available to GEM for this step, seven old events and seven new ones. These events and the seven rules induced during the first step were used to form new rules.

At each step in the incremental learning process four rule types were formed:

1. A control set of discriminant rules formed using the single-step version of AQ.
2. A set of discriminant rules formed incrementally.
3. A set of characteristic rules formed incrementally.

| Class | Generated Number | Available Events | Events to be Added | Total Events for this learning step |
|---|---|---|---|---|
| *Stenonema carolina* | 0.32 | 4 | 1 | 2 |
| *Stenonema candidum* | 0.53 | 4 | 2 | 3 |
| *Stenonema floridense* | 0.21 | 4 | 0 | 1 |
| *Stenonema gildersleevei* | 0.06 | 4 | 0 | 1 |
| *Stenonema interpuc* | 0.89 | 4 | 3 | 4 |
| *Stenonema minnentonka* | 0.43 | 4 | 1 | 2 |
| *Stenonema pallidum* | 0.11 | 4 | 0 | 1 |
| Total | – | 28 | 7 | 14 |

Figure 3. Event selection for the second learning step in the *Stenonema* mayfly nymph domain.

4. A set of discriminant rules induced from the characteristic rules, above.

All three discriminant rule sets were tested against all available testing events. In each domain, the entire experiment was repeated with different combinations of learning and testing events. The results of these experiments are summarized in the next section.

## 5. EXPERIMENTAL RESULTS

Three facets of rule induction were measured. First, the rule induction time was estimated based on the c.p.u. time used by GEM in forming the rules. All results are for a Pascal version of the GEM program running under the 4.2 bsd version of the UNIX operating system on a VAX 11/780. Second, rule comprehensibility was measured. A rule's complexity, assumed to be the inverse of its comprehensibility, was defined as the sum of the number of selectors, number of different attributes and number of complexes in the rule. The complexity of a set of rules is the average of the complexities of the members. Third, the performance of the rules was estimated. Rules were tested using the ATEST program and testing examples set aside for the purpose [see Reinke (1984) for a description of ATEST and a discussion of the issues involved in rule evaluation].

**5.1. Mayfly nymph extension**

Figure 4 shows the c.p.u. time used by GEM to induce three different types of discriminant rules for identifying *Stenonema* mayfly nymphs. As expected, inducing DD descriptions from CD descriptions took very little time (less than 1 s of c.p.u. time in every case). The incremental method created descriptions in considerably less time than the single-step method.

Figure 5 shows the complexity of all four rule types at each stage of the learning process. The complexity of the discriminant rules induced incrementally rose at every step, undoubtedly due to the specialization of complexes. There was little difference between the characteristic rules and the discriminant rules induced from them. The second repetition of the experiment, using different learning events, produced more complex characteristic rules and simpler discriminant ones.

The performance of the three discriminant rule types is compared in Figure 6. In this domain, almost all misclassifications took place because several descriptions were satisfied by a testing event. Therefore, the DD

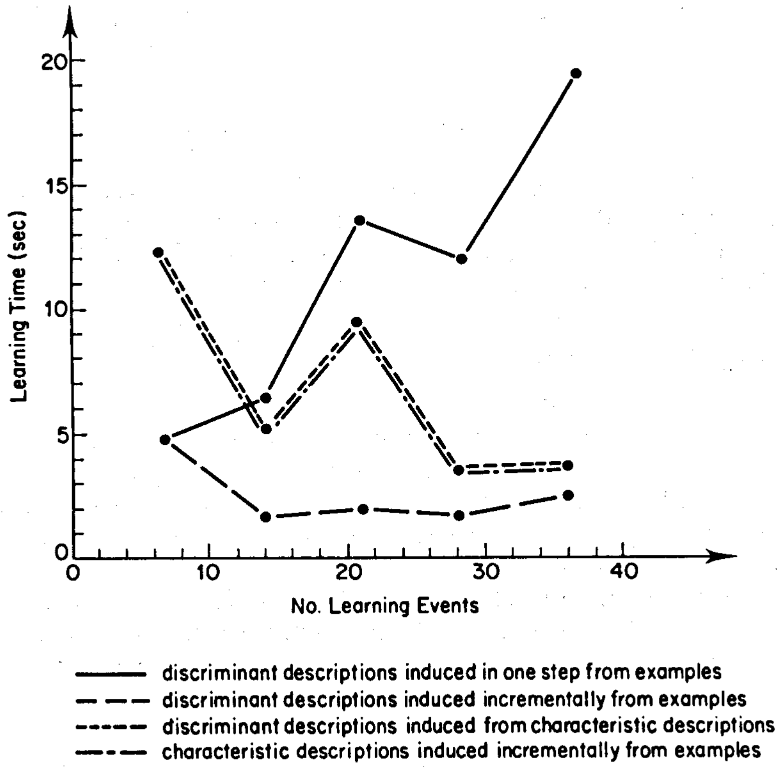


Figure 4. c.p.u. time to induce four description types for identification of mayfly nymphs.

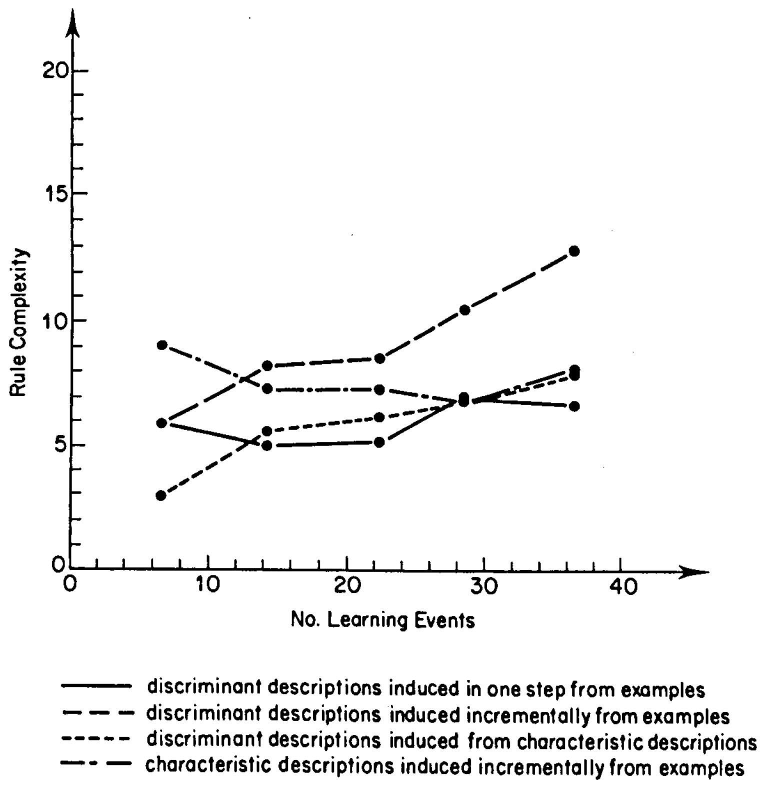


Figure 5. Complexity of four description types for identification of mayfly nymphs.

descriptions induced from CD descriptions were too general in the tests shown in Figure 6. A repetition of the experiment produced CD descriptions from which better DD descriptions were induced. Typical descriptions in this domain are shown in Figure 7.

## 5.2. Chess endgame position classification

In the chess endgame problem area, it was not possible to generalize the characteristic descriptions produced by GEM. For this reason, Figures 8–10 compare the two types of discriminant rules and the characteristic rules. Figure 8 shows that the incremental method saved a considerable amount of induction time in this domain.

Figure 9 compares the complexity of the three rule types over the course of the learning process. Characteristic and discriminant descriptions differed very little overall. This, and GEM's inability to generalize the long descriptions, is probably due to the nature of the attributes used to describe events. Since each input vector is really a generalization of several actual chess positions, one event may not generalize easily to

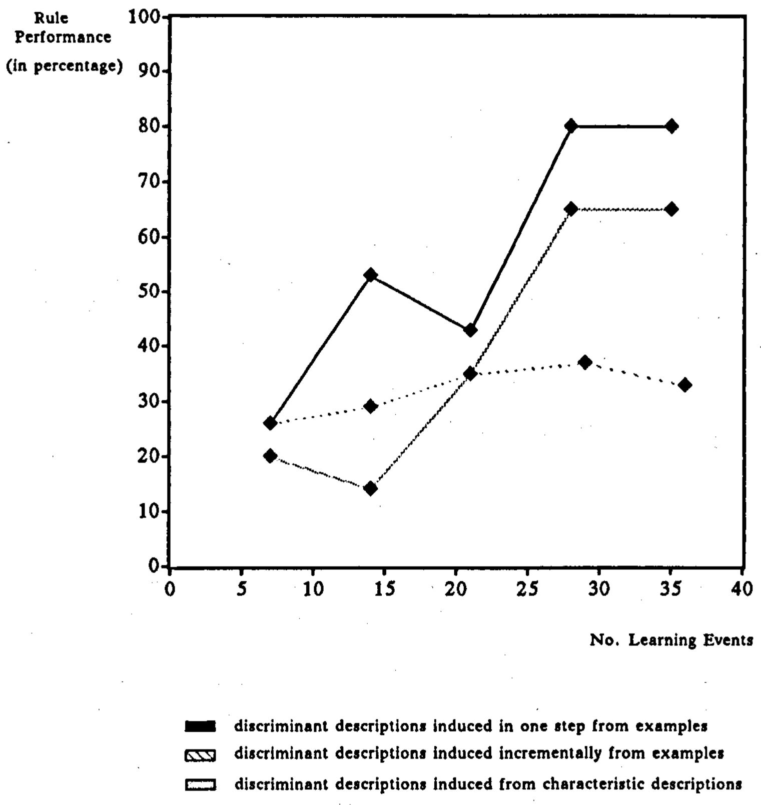


Figure 6. Performance of three description types for identification of mayfly nymphs.

cover another. This hypothesis is partially borne out by the fact that the descriptions produced were very disjunctive, containing an average of 20 complexes each. Each of these complexes would be highly specialized (i.e. characteristic) by nature, and therefore impossible to generalize.

Figure 10 shows the performance of all three description types. Unsurprisingly, the choice of learning events was very important in this domain. Two rules sets were produced by induction over two learning sets of exactly the same size, yet the rules were more than 90 per cent correct during the run shown and about 50 per cent correct during the other. This suggests that events of a given class appear in many distinct regions of the event space, and explains the highly disjunctive nature of descriptions in this domain. If learning events are taken from only a few of the regions, then rule performance will be poor. If, however, the learning events contain elements from almost all the regions, the rules should have relatively good performance. This hypothesis suggests that the better rules should have a larger number of complexes than the

Characteristic description:

[maxilla_crown_spines = 10][maxilla_lateral_setae = 21,26,28,30][inner_canine_teeth = 2]
[outer_canine_teeth = 7.8][terga_dark_posterior_margins = absent]

Discriminant description induced from characteristic description:

[maxilla_crown_spines = 10][inner_canine_teeth = 2][terga_dark_posterior_margins = absent]

Discriminant description induced from examples:

[terga_mid_dorsal_pale_streaks = absent]

Figure 7. Examples of different description types for the class *Stenonema carolina* in the mayfly nymph domain.

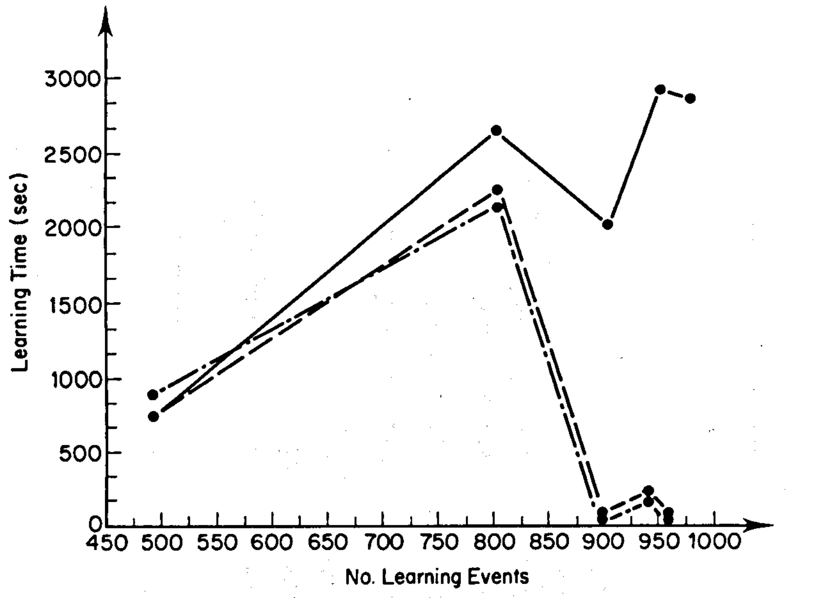


Figure 8. c.p.u. time to induce three description types for classification of KPK endgame positions.

poorer. This was indeed the case—the good rules had, on the average, almost twice as many complexes as the poorer rules. It should be noted that this effect would probably not have been observed if a chess expert had chosen the examples. Typical descriptions for this domain are shown in Figures 11 and 12.

**5.3. Soybean disease diagnosis**

The results for the soybean disease problem are summarized in Figures 13–15. The event space for this problem was by far the largest of the three, so rule induction took considerably longer. The time saved by using the incremental method was considerable. Again, inducing DD descriptions from CD descriptions took very little time.

Figure 14 shows the complexity of the four description types over the learning process. As expected, the characteristic descriptions were the most complex. The DD descriptions induced from CD descriptions were more complex than DD descriptions induced directly from examples.

All of the discriminant rules performed well, as shown in Figure 13. In comparison, the most recent rules written by plant pathologists were

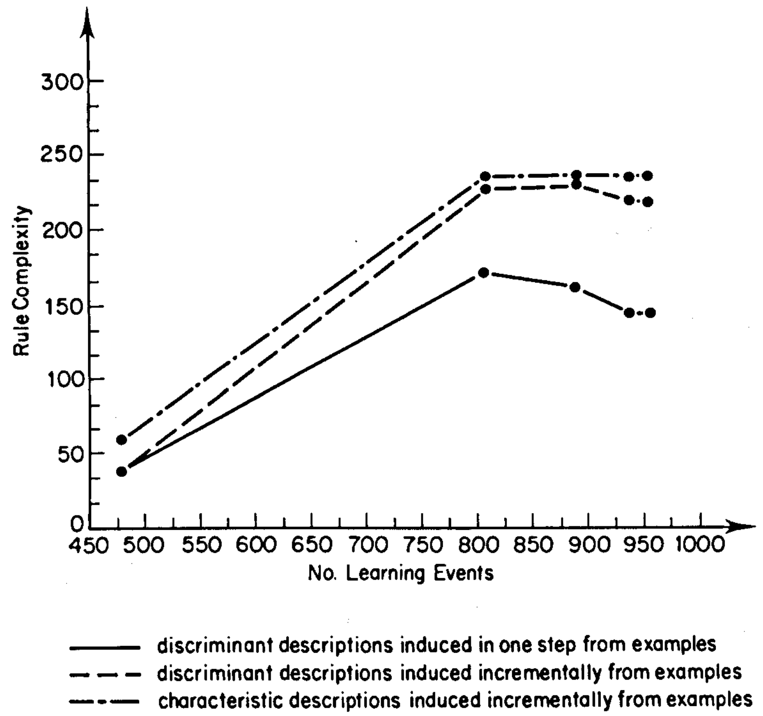


Figure 9. Complexity of three description types for classification of KPK endgame positions.

about 80% correct for these testing events. These results are similar to earlier results in the same domain (Michalski, 1980b; Niblett, 1982). Typical descriptions for this domain are shown in Figure 16.

## 6. SUMMARY

The experimental results are summarized below in terms of the goals sets forth in Section 4:

1. The relative quality of the various description types varied widely with the domain. In the mayfly nymph recognition domain, the incrementally learned descriptions performed poorly compared to the single step descriptions (83% correct compared to 60%). In the chess endgame domain they performed at about the same level (98–96 per cent) and in the soybean disease diagnosis domain the incrementally learned rules performed slightly better (88–82 per cent). Overall, incrementally learned discriminant disjunctive descriptions were slightly more complex than descriptions formed in a single step. Characteristic disjunctive descriptions were even more complex, as expected, but were unfortunately also more disjunctive (averaging six complexes per descrip-

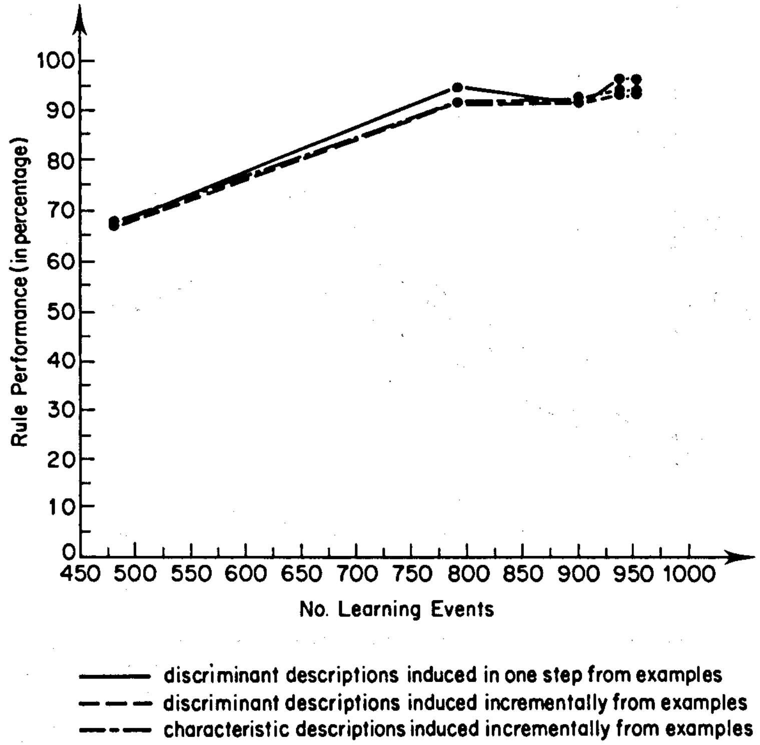


Figure 10. Performance of three description types for classification of KPK endgame positions.

tion over the three domains compared to four complexes per description for discriminant disjunctive).

2. The discriminant disjunctive descriptions formed from characteristic disjunctive descriptions performed better than the discriminant disjunctive descriptions learned from examples in two of the three domains. Overall, the performance of these descriptions was about four per cent better than that of the discriminant disjunctive descriptions induced from examples. Unfortunately, inducing discriminant disjunctive descriptions from characteristic disjunctive makes the discriminant disjunctive description more complex (the average complexity of indirectly induced descriptions was 58, compared to 41 for descriptions induced directly from examples).

3. Both incremental methods were significantly faster than single step learning (between two and five times as fast overall). Summing over all experiments in all domains, the single step method took approximately $4.2 \times 10^3$ c.p.u. minutes, the incremental method took $0.7 \times 10^3$ c.p.u. min and the characteristic disjunctive to discriminant disjunctive incremental method took $2.6 \times 10^3$ c.p.u. min.

```
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][rstal = f]V
[cimmt = f][mmp2 = t][btop5 = f][spra7 = f][srfil = f]V
[cimmt = f][cplu2 = f][cplu1 = 1][cahea = f][rneac = f][rrp2 = f][rsta] = f]V
[cimmt = f][rrp2 = f][mp5 = t][spra7 = f][spran = t][srfil = f]V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rnear = f][rstal = f][mpmov = t][spra7 = f][srfil = t]V
[cimmt = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][rstal = f]
[mmp2 = t][srfil = t][nxto7 = t]V
[cimmt = f][cahea = f][rnear = f][btop5 = f][spra7 = t][srfil = f][nxto7 = t]V
[cimmt = f][ccrit = f][mmp1 = t][mp5 = t][spra7 = f][smain = t][srfil = f]V
[cimmt = f][rrp2 = f][mpmov = t][mp5 = f][spra7 = f][smain = t][srfil = f]V
[cimmt = f][ccrit = f][btop5 = f][spra7 = t][srfil = f][nxto7 = t]V
[cimmt = f][btop5 = f][spra7 = f][spran = 7][srfil = f]V
[cimmt = f][ccrit = f][mdiro = t][btop5 = f][spra7 = f][srfil = f]V
[cimmt = f][ccrit = f][spra7 = f][smain = t][srfil = f][sint = t]V
[cimmt = f][ccrit = f][diro5 = t][spra7 = t][spra7 = f][spran = t][srfil = f]V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rstal = f][mp5 = f][srfil = t][nxto7 = t]V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][rstal = f][mmp1 = t][mp5 = t][spra7 = f]V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rstal = f][mmp2 = t][mp5 = t][spra7 = f]V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][rrp1 = t][rstal = f][spra7 = f][srfil = t]V
[cimmt = f][cplu1 = f][cwksa = f][rrp2 = f][rstal = f][mpmov = t][mp5 = f][spra7 = f][srfil = t]V
[cimmt = f][rrp2 = f][r5p6 = t][spran = t][srfil = f][nxto7 = t]V
[cimmt = f][ccrit = f][mpmov = t][spra7 = f][smain = t][srfil = f]V
[cimmt = f][ccrit = t][rneac = f][rrp2 = f][rstal = f][mmp2 = t][mpmov = t][spra7 = f][srfil = t]V
[cimmt = f][cahea = f][ccrit = t][rrp2 = f][rnear = f][mp5 = t][srfil = f][nxto7 = t]V
[cimmt = f][rneac = f][rstal = f][mmp2 = t][btop5 = f][spra7 = f]
```

Figure 11. Typical character description for the class *Won for white* in the KPK chess endgame domain.

```
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rpsq = f][rrp2 = f]V
[cimmt = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f[[mmp2 = t][nxto7 = t]V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][mpmov = t]V
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][mp5 = t][nxto7 = t]V
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][srfil = t]V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rnear = f][mmp1 = t][mp5 = t]V
[cimmt = f][cplu1 = f][cwksa = f][rneac = f][rrp1 = t][rrp2 = f]V
[cimmt = f][rnea8 = f][rrp2 = f][rnear = f][mpmov = t][smain = t]V
[cimmt = f][cplu2 = f][cplu1 = f][rrp2 = f][mpmov = t][smain = t]V
[cimmt = f][rneac = f][mmp2 = t][btop5 = f][nxto7 = t]V
[cimmt = f][cwksa = t][rneac = f][mmp1 = t][btop5 = f]V
[cimmt = f][cwksa = f][rrp2 = f][mp5 = t][spran = t]V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][rnear = f][mp5 = t][spra7 = t][nxto7 = t]V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = f][mp5 = t][spra7 = t]V
[cimmt = f][cwksa = f[btop5 = f][spran = t]V
[cimmt = f][rrp2 = f][mmp2 = t][mp5 = t][smain = t]V
[cimmt = f][cplu1 = f][rrp2 = f][rnear = f][mdiro = t][mp5 = t][smain = t]V
[cimmt = f][cplu1 = f][ccrit = f][smain = t][sint = t]V
[cimmt = f][cwksa = f][rrp2 = f][diro5 = t][spran = t]V
[cimmt = f][cwksa = f][ccrit = t][rnea8 = f][rrp2 = f][rnear = f][mpmov = t][mp5 = f][srfil = t]V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rnear = f][mp5 = f][srfil = t]V
[cimmt = f][cplu1 = f][cwksa = t][rrp2 = f][mpmov = t][smain = t]V
[cimmt = f][cplu1 = f][cahea = t][rneac = f][rrp2 = f][mmp2 = t][mp5 = t]V
[cimmt = f][ccrit = f][mmp1 = t][mp5 = t][smain = t]V
[cimmt = f][cplu1 = t][rrp2 = f][r5p6 = t][spran = t]V
[cimmt = f][ccrit = f][rneac = f][mpmov = t][smain = t]V
[cimmt = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rnear = f][mp5 = t][spra7 = t][nxto7 = t]V
[cimmt = f][cplu1 = t][rrp2 = f][mp5 = t][spran = t]V
[cimmt = f][cplu1 = t][ccrit = t][rneac = f][rrp2 = f][rnear = f][mpmov = t][srfil = t]V
[cimmt = f][cahea = t][ccrit = f][mdiro = t][btop5 = f][smain = t]
```

Figure 12. Typical discriminant description for the class *Won for white* in the KPK chess endgame domain.

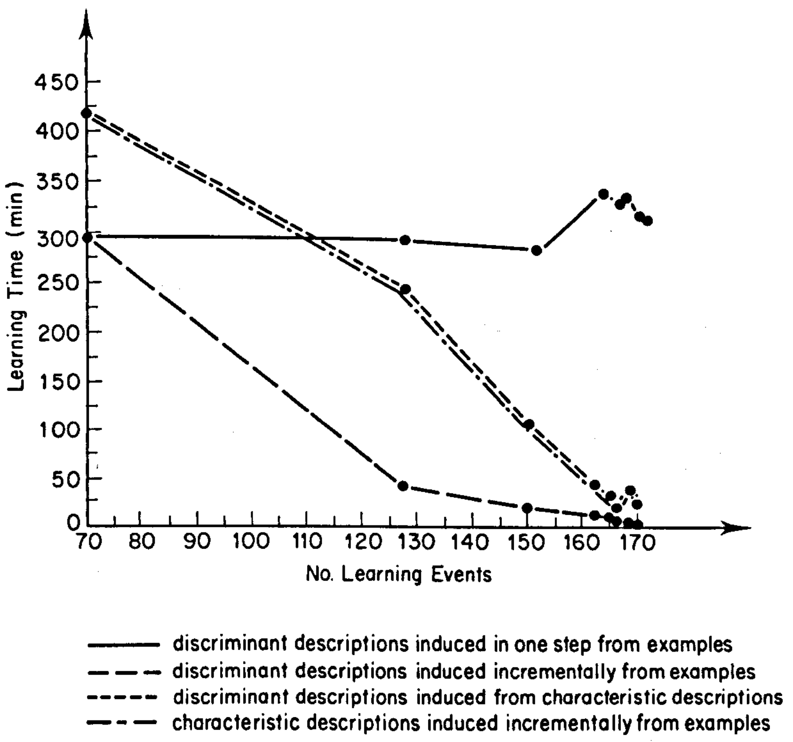


Figure 13. c.p.u. time to induce three description types for soybean disease diagnosis.

The success of the full memory incremental learning method was obvious. In all the application areas, GEM took considerably less time to form rules when it had old rules to modify. The rules produced using the incremental method were slightly more complex and performed slightly less well than those produced in a single step, but the time saved was large and the differences in performance and complexity were small. The method of inducing discriminant disjunctive descriptions from characteristic disjunctive descriptions proved workable, but produced more complex rules. This may have been due to the nature of the characteristic descriptions produced by GEM.

The incremental method could be further enhanced by simplifying both the specialization and generalization steps using the reunion operator. That is, a complex could be simplified by taking the union of the events it covers. New positive events could be covered by taking the union of the events and some complex. The method currently used could serve as a back-up, invoked only when reunion produces a complex which does not satisfy specified constraints.

The characteristic descriptions produced by GEM were sometimes unattractive because they were long and disjunctive. A combination of

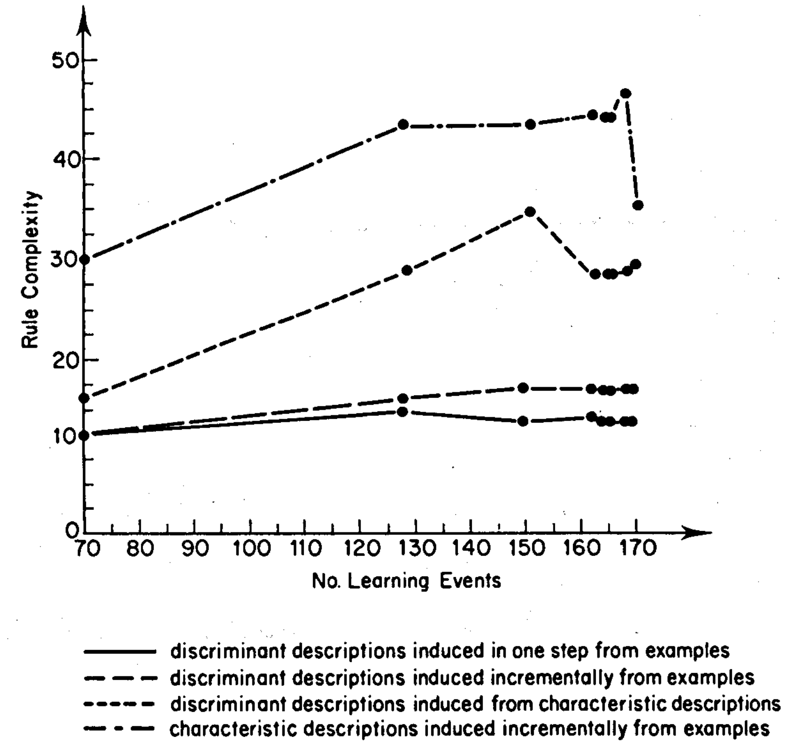


Figure 14. Complexity of four description types for soybean disease diagnosis.

two factors was responsible: the individual concepts in each domain tended to be divided into subparts and GEM always produces consistent *and* complete descriptions. Formally (see Section 2), characteristic descriptions are not necessarily consistent. Nevertheless, discriminant descriptions induced from these characteristic descriptions were often quite good.

A simple method could be used to produce conjunctive descriptions. If the disjuncts in a characteristic disjunctive concept description produced by GEM correspond to subclasses, a tree-structured concept description could be formed in the following way:

1. Induce a characteristic disjunctive description incrementally from examples.

2. Treat each disjunct as a separate class within the concept and induce a description to characterize each subclass.

Another possibility is to use a conceptual clustering method such as that described by Michalski and Stepp (1983) to divide each class into subclasses.

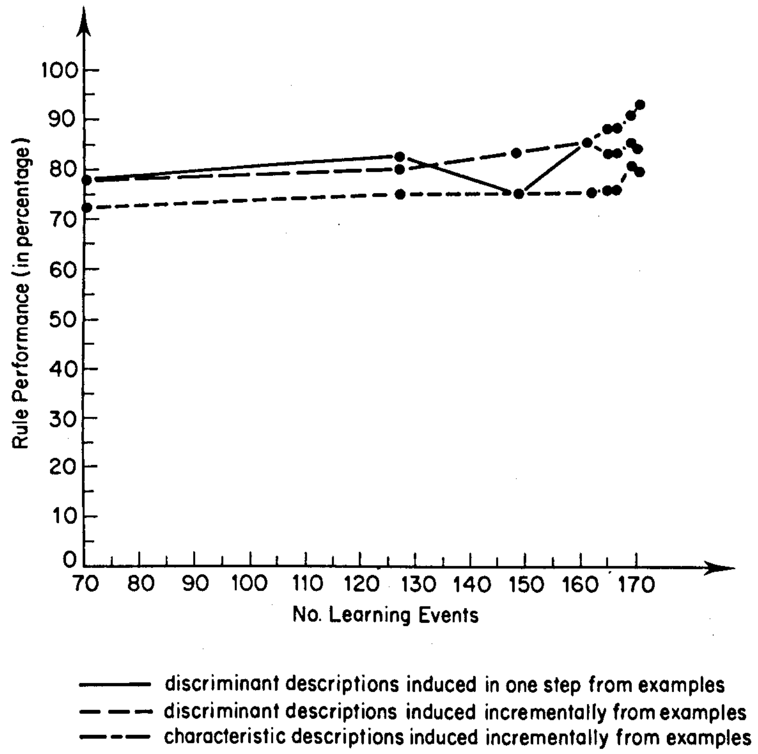


Figure 15. Performance of three description types for soybean disease diagnosis.

A more difficult extension to the method would use *partial memory* and *exceptions* (Michalski and Winston, in press). A partial memory incremental learning system would have to be able to recognize and remember 'important' events. Something like this is done by ID3 (Quinlan, 1979), which remembers one event in each parcel of events that contributed to rule formation. A true partial memory incremental learning system will need some criteria recognizing importance. Exception events which violate the consistency of conjunctive characteristic descriptions are interesting candidates. The method would have to form a characteristic conjunctive description while creating as few exception events as possible.

Unless a data base of examples is excessively large, the full memory incremental learning method provides the best way to induce reliable concept descriptions. For three real world problems, the full memory method took considerably less time and no more memory than the single step method (which must have all the events in memory anyway). Further, it appears that the best way to learn incrementally is to maintain

Characteristic description:

[precipitation = above_normal][temperature = normal . . . above_norm][severity = minor . . . potentially_severe]
[condition_of_leaves = abnormal][leaf_spot_color = brown]
[leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins][leaf_spot_size = greater_than_eighth_inch]
[shot_holing = present][position_of_affected_leaves = scattered_on_plant]
[condition_of_leaves_below_affected_leaves = unaffected][stem_cankers = does_not_apply][fruit_spots = colored_spots]

Discriminant description induced from characteristic description:

[leaf_spot_color = brown][leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins]
[position_of_affected_leaves = scattered_on_plant][fruit_spots = colored_spots]

Discriminant description induced from examples:

[leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins]

Description written by domain expert;

[leaf_spot_growth = scattered_with_concentric_rings]:0.90
+
[time_of_occurrence = august . . . october][shot_holing = present]:0.50
+
[leaf_spot_size = greater_than_eighth_inch]:0.45
+
[time_of_occurence = august . . . october][fruit_pods = diseased][fruit_spots = colored_spots]:0.10
+
[seed_discoloration_color = black]:0.05
+
[leaf_spot_margins = water_soaked]:0.05
+
[yellow_leaf_spot_halos = absent]:0.05

Figure 16. Examples of different description types for the class *Alternaria Leaf Spot* in the soybean disease diagnosis domain.

characteristic descriptions of classes. Such descriptions are more appealing to humans than terse, disjunctive descriptions. The results here show that characteristic type descriptions also contain enough information that good discriminant descriptions may be induced from them in a very short amount of time.

**Acknowledgements**

The first phase of this research was done at the University of Illinois, where it was supported in part by the National Science Foundation under grant DCR 84-06801, and in part by the Office of Naval Research under grant N00014-82-K-0186. Subsequent research was done while the first author was at GTE Laboratories and the second author was at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Support for the Artificial Intelligence Laboratory's research is provided in part by the Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-80-C-0505. The authors would like to thank A. Paterson for providing the chess endgame data, and Dr D. Prerau and D. Laukaitis for reading drafts of this paper.

## REFERENCES

Becker, J. (1985) Topics in incremental learning of discriminant descriptions, *UIUCDCS-F-*85-935. Department of Computer Science, University of Illinois at Urbana-Champaign.

Hong, J. R. and Michalski, R. S. (1985) The general covering problem: an extension matrix method for generating disjoint stars. Report of the Intelligent Systems Group. Department of Computer Science, University of Illinois at Urbana-Champaign.

Langley, P., Bradshaw, G., and Simon, H. A. (1983) Rediscovering chemistry with the BACON System. In *Machine learning, an artificial intelligence approach* (eds R. S. Michalski, J. B. Carbonell and T. Mitchell) pp. 307–330. Tioga, Palo Alto.

Lewis, P. (1974) Taxonomy and ecology of *Stenonema* mayflies (Heptageniidae: Epheme-roptera), *EPA*-670*A*-74-006. Environmental Protection Agency, Washington DC.

Michalski, R. S. (1969) On the quasi-minimal solution of the general covering problem. *Proc. 5th Int. Symp. on Information Processing* (*FCIP* 69), Vol. 13, pp. 125–127, Bled, Yugoslavia.

Michalski, R. S. (1975) Variable-valued logic and its applications to pattern recognition and machine learning. In *Computer science and multiple-valued logic theory and applications* (ed. D. C. Rine) pp. 506–534. North-Holland, Amsterdam.

Michalski, R. S. (1980a) Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2**, 349–361.

Michalski, R. S. (1980b) Knowledge acquisition through conceptual clustering: a theoretical framework and an algorithm for partitioning data into conjunctive concepts. *Int. J. Policy Analysis and Information Systems* 4(3), 219–44.

Michalski, R. S. (1983) A theory and methodology of inductive learning. In *Machine learning, an artificial intelligence approach* (eds R. S. Michalski, J. B. Carbonell and T. Mitchell, pp. 83–124. Tioga, Palo Alto.

Michalski, R. S. and Larson, J. B. (1978) Selection of most representative training examples and incremental generation of $VL_1$ hypotheses: the underlying methodology and descriptions of programs ESEL and AQ11, *Report* 867. Department of Computer Science, University of Illinois at Urbana-Champaign.

Michalski, R. S. and Chilausky, R. L. (1980) Learning by being told and learning from examples: an experimental comparison of two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Int. J. Policy Analysis and Information Systems* 4(2), 125–160.

Michalski, R. S. and Stepp, R. E. (1982) Revealing conceptual structure in data by inductive inference. *Machine intelligence* 10 (eds J. E. Hayes, D. Michie & Y.-H. Pao) pp. 173–195. Ellis Horwood, Chichester; Halsted, New York.

Michalski, R. S. and Dietterich, T. G. (1983) A comparative review of selected methods for learning from examples. *Machine learning, an artificial intelligence approach* (eds R. S. Michalski, J. B. Carbonell and T. Mitchell) pp. 41–75. Tioga, Palo Alto.

Michalski, R. S. and Stepp, R. E. (1983) Learning from observation: conceptual clustering. *Machine learning, an artificial intelligence approach,* (eds R. S. Michalski, J. B. Carbonell and T. Mitchell) pp. 331–363. Tioga, Palo Alto.

Michalski, R. S. and Winston, P. H. (1985) Variable precision logic, *AI Memo 857,* Cambridge: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

Michalski, R. S., Davis, J. H., Bisht, V. S., and Sinclair, J. B. (1983) A computer-based advisory system for diagnosing soybean diseases in Illinois. *Plant Disease* 67(4), 459–63.

Niblett, T. B. (1982) A provably correct advice strategy for the endgame of king and pawn versus king. *Machine intelligence* 10 (eds J. E. Hayes, D. Michie and Y.-H. Pao) pp. 101–122. Ellis Horwood, Chichester/Wiley, New York.

Quinlan, J. R. (1979) Discovering rules by induction from large collections of examples. *Expert systems in the micro electronic age* (ed. D. Michie) pp. 168–201. Edinburgh University Press, Edinburgh.

Quinlan, J. R. (1982) Semi-autonomous acquisition of pattern based knowledge. *Machine Intelligence* 10 (eds J. E. Hayes, D. Michie and Y.-H. Pao) pp. 159–172. Ellis Horwood, Chichester/Wiley, New York.

Reinke, R. E. (1984) Knowledge acquisition and refinement tools for the ADVISE meta-expert system, *UIUCDCS-F*-84-921. Department of Computer Science, University of Illinois at Urbana-Champaign.

Spackman, K. A. (1983) QUIN: integration of inferential operators within a relational data base, *UIUCDCS-F*-83-917. Department of Computer Science, University of Illinois at Urbana-Champaign.

Winston, P. H. (1975) Learning structural descriptions from examples. In *The psychology of computer vision* (ed. P. H. Winston). McGraw-Hill, New York.

12

# Generating Expert Rules from Examples in PROLOG

B. Arbab*
Department of Computer Science,
University of California at Los Angeles, USA
D. Michie
The Turing Institute,
Glasgow, UK

**Abstract**
Automatic decision-tree generators have been used in the production of expert systems. A number of experiments indicate the need for more linear (and hence more understandable) yet efficient decision trees. An algorithm is implemented for constructing decision trees optimized with respect to linearity. It also improves on a previous linearizing algorithm (AOCDL) with respect to execution efficiency.

## 1. PROBLEM DESCRIPTION AND ASSUMPTIONS

This work describes tools for generating decision-trees that are optimized with respect to linearity and are more efficient than those generated by Bratko's AOCDL [1]. The rule generator is specialized so as to obey stated constraints corresponding to the above two properties. Rule induction takes advantage of one of the expert's most reliable and highly developed skills [2], teaching by example, and this avoids the need to resort to dialogue-acquisition of rules, traditionally recognized as the bottle-neck problem of knowledge engineering. However, decision-trees derived from situation-action pairs are inherently less descriptive for expressing concepts than first-order or multi-valued logic used in other projects [3–5]. The lack of descriptive power is primarily associated with the absence of quantified variables.

Quinlan [6] and Shapiro [7] have demonstrated that generation of decision-trees from a set of examples provided by a domain expert is a practical method for knowledge acquisition (see also Martelli and Montanari [8] for generation of optimal trees). This paper stems from two previous approaches to the rule-induction problem: (i) Quinlan's ID3 uses an information theoretic approach to control a 'best-first no

* Present address: IBM Los Angeles Scientific Center, 11601 Wilshire Boulevard, Los Angeles CA 90025-1738.

backtrack' search, producing decision-trees of high, but not optimal, execution efficiency; (ii) Bratko's AOCDL uses backtrack heuristic search. ID3 ignores the human understandability criterion for induced rules while AOCDL ignores the efficiency criterion. ID3's attribute selection criterion, based on entropy, promotes efficient decision-tree execution on the machine. However, the decision-trees are not easily understood by humans. AOCDL is heuristically guided to minimize a non-linearity (branching) measure. Arbitrarily branching structures are hard for a human to keep mental track of. So one idea is to only allow for linear or almost linear decision-trees [9]. A decision-tree is said to be linear if every node has at most one non-terminal son. Note that even trees with high branching ratios (multiple-value attributes) and multiple decision classes can be linear. The relation between linear trees and understandability has been experimentally investigated by Shapiro and Niblett [7, 10]. In two separate classification tasks in chess end-games, structured representations with tree-linearity constraint were uniformly understandable, whereas representations in the form of arbitrarily branching decision trees were uniformly opaque. Our Rule Generator (RG), which was implemented in PROLOG produces decision-trees that are linear where such trees exist. In cases where such a tree does not exist, the most linear tree is constructed. The derived trees are efficient at execution time. These two requirements, linearity and efficiency, are inversely related. A balanced tree is shallower and more efficient for machine execution than a linear tree. In synthesizing decision-trees, however, we always trade efficiency for linearity, in much the same way that structured programming trades efficiency for program clarity and readability.

The presence of a domain expert makes 'structured induction' possible, which breaks the problem into subproblems. A detailed description of structured induction is given by Shapiro and Niblett [7,10]. With structured induction the size of the example sets is never large, e.g. at most in the order of tens. It has been found by Quinlan [6] that small example sets are sufficient to generate rules capable of classifying even large domains with high reliability.

We have developed RG under the assumptions that:

1. Structured induction is feasible.

2. Linearity of decision-trees is to be optimized even at the expense of efficiency.

3. Efficiency of decision-trees is to be increased only subject to the constraint that linearity is *not* affected.

The decision-trees in Figure 1 correspond to an example set taken from a planning domain for building an arch, see [4]. Each node of the tree corresponds to an attribute, leaf nodes represent decision classes, and the labels on the arcs are the attribute's values. These trees were induced by: (1) Expert-Ease [12], a commercial version of the ID3-derived ACLS

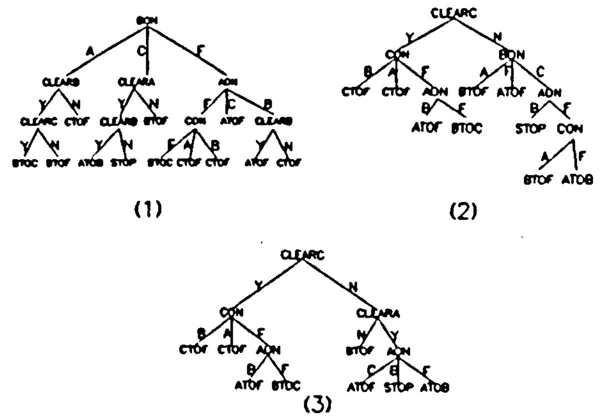


Figure 1. Trees with different linearity and efficiency.

algorithm which produces efficient but non-linear trees; (2) AOCDL, which maximizes linearity but not efficiency; and (3) RG, which maximizes linearity and promotes efficiency. The exact non-linearity and efficiency (execution cost) measures of these trees can be seen for comparison in Figure 10 under EX4.

Tree (1) has eight non-terminals (attributes) and 12 terminal (class) nodes, while tree (3) has five attributes and eight classes. Tree (1) has an irregular branching structure, while tree (3) has a small non-linearity that occurs at the root. Decision trees can be used to classify new examples. An interpreter for executing such trees will compute a value for an attribute (non-terminal). This value specifies which branch of the tree should be traversed next. A classification has been made when a class (terminal) node is reached.

RG incorporates linearity and efficiency measures within an AO* [13] algorithm as heuristics to guarantee optimal linearity. The efficiency of the decision-trees is increased according to each candidate attribute's expected information contribution if appended at the given point in the tree, i.e. attributes with high information content will be placed as high in the decision tree as possible thus increasing the probability of classification to occur as early as possible.

The following points should be read against the background of examples such as those worked in the Appendix. They concern theoretical limitations of the rule synthesis process and how this process may be used for solving problems. These three properties of induction define, more precisely, what can be expected from the rule synthesis process.

1. Any rule synthesized from only a subset of all possible examples, cannot in general be proved correct in all cases (i.e. all possible examples). However, this does not imply that correct rules for all possible cases cannot be constructed from a minimal set of examples.

2. In applications where correct classification is required of finite sets of examples taken from a finite domain, one can consider a table look-up process. It may still be desirable to use rule synthesis in such cases since large tabulations can be condensed, without loss of information or accuracy, into simple rules.

3. The synthesis process cannot by itself find any hidden hierarchical structures in example sets. In structured induction, these structures are thought out by the expert and documented before the rule induction process is started. A structure set by the expert can later be modified as the project progresses.

The generated rules are semantically equivalent to the example sets they originated from when these are complete, i.e. no information has been added or lost by the rule-generating program. It is this characteristic of rule-generating programs that sets them apart from generalization techniques used in other projects such as Marvin [14] or MIS [15].

## 2. DECIDER STATUS OF ATTRIBUTES

Experts are generally adept at communicating their expertise by means of examples. The examples thus form a language through which knowledge is communicated. There are three parts to this language: attributes, classes, and examples, the latter being defined in terms of attribute values and classes. Attributes are defined by the domain specialist as critical features or relevant facts, e.g. colour of eyes with values brown, black, and blue; or kidney size with values large, small, and normal. Class values correspond to the identification made on the basis of attribute values, e.g. normal, or kidney stone. Examples are represented as attribute-value vectors paired with classes as shown in Figure 2. One additional criterion imposed on examples is that they must be clash free. A clash is defined as two identical vectors leading to different class values. Clashes usually signify the inadequacy of attributes

| A1 | A2 | A3 | A4 | A5 | class |
|---|---|---|---|---|---|
| t | t | f | t | t | c1 |
| t | f | t | f | f | c1 |
| t | f | f | t | f | c1 |
| f | t | f | t | t | c2 |
| t | t | f | t | f | c2 |
| t | f | t | t | f | c2 |
| f | f | t | f | t | c1 |
| t | f | f | t | t | c1 |
| t | t | t | t | f | c2 |
| f | f | f | f | f | c1 |
| f | t | f | f | t | c2 |

Figure 2. An example set.

for classification of the problem and can be removed by introducing additional attributes.

With respect to a specified example set, an attribute has a decider status: total, partial, or non-decider. 'Decisiveness' of an attribute may be computed from a matrix whose rows correspond to class values and columns to values the attribute of interest may range over. Each entry in this matrix corresponds to a frequency count of the class per attribute's value. An attribute's decider status is defined as follows:

1. Total Decider, if the attribute partitions the example set such that each partition belongs to a single class. In matrix form this corresponds to the condition that there is at most one non-zero value in each column.
2. Partial Decider, if the attribute partitions the example set such that all but one partition belong to a single class. In matrix form this corresponds to at least one column where all entries except one are zero.
3. Non-Decider, if neither of the above is true.

In Figure 3 we show an example set and the matrices corresponding to each attribute's decisiveness status. Attribute *A*1 is a total decider, *A*2 a partial decider and *A*3 a non-decider.

Decider status of an attribute plays an important role in our search for linear decision-trees. If an attribute must be selected from a set of total or partial deciders, then the linearity of the final tree is not affected by the choice of a particular attribute, but efficiency can depend on this choice. However, selection of a non-decider attribute can affect efficiency and invariably destroys linearity. Consider the example set in Figure 2 where for simplicity we have assumed binary attributes and only two classes.

It so happens that in the above example set all candidate attributes for the top of the tree are non-deciders. However, different attributes lead to various non-linear trees. Using attribute *A*2 at the top leads to a tree of the form shown on the left side of Figure 4 while using any of attributes *A*1, *A*3, *A*4, or *A*5 leads to a tree of the form shown on the right.

Clearly the tree on the right is a more linear tree (we recall that linear decision-trees are easier to understand). Thus, when selecting an attribute from a set of non-deciders one must consider their effect on the

| A1 | A2 | A3 | Class |
|---|---|---|---|
| t | t | t | c1 |
| t | t | f | c1 |
| f | f | f | c2 |
| f | t | t | c2 |

| A1 | f | t |
|---|---|---|
| c1 | 0 | 2 |
| c2 | 2 | 0 |

| A2 | t | f |
|---|---|---|
| c1 | 2 | 0 |
| c2 | 1 | 1 |

| A3 | t | f |
|---|---|---|
| c1 | 1 | 1 |
| c2 | 1 | 1 |

Figure 3. computation of decider status.

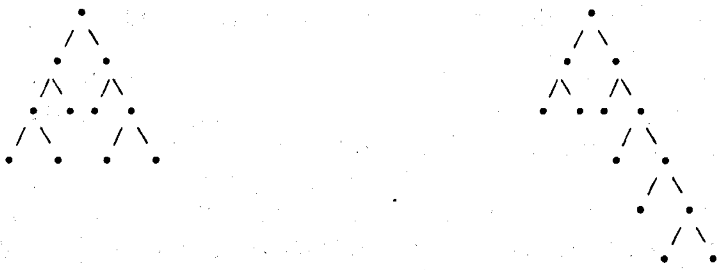


Figure 4. Trees with different linearity measures.

overall linearity and efficiency of the decision-tree. In general, selecting the right attribute requires a search procedure which is described in later sections.

## 3. LINEARITY MEASURE

Degree of linearity has been used as a measure of desirability for trees. This concept must be formalized to allow comparison of trees on the basis of their non-linearity. Some desirable characteristics of a function to compute non-linearity of trees are:

1. An intuitive, yet formal, basis.
2. Sensitivity to the size of trees.
3. Sensitivity to location of non-linearity in a tree.

Bratko [1] has proposed such a function. The non-linearity measure which he proposes is based on the fact that traversal of a linear tree requires scanning through contiguous memory locations and minimizes jumps across the memory. This may be one reason why linear decision-trees are easier for humans to understand than are non-linear trees. Let $T$ be a decision tree whose root is $A$ and subtrees are $S1, S2, \ldots, Sm$, as in Figure 5.

The proposed measure for non-linearity is:

$$NL(T) = (1/m) \times \sum_{i=1}^{m} [NL(Si) + (m - i) \times s(Si)]$$

where $NL(T)$ denotes the non-linearity of $T$, $NL(T) = 0$ when $T$ is a class value (leaf node) and the number of internal nodes of the tree, $s(T)$ is

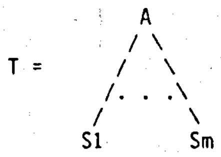
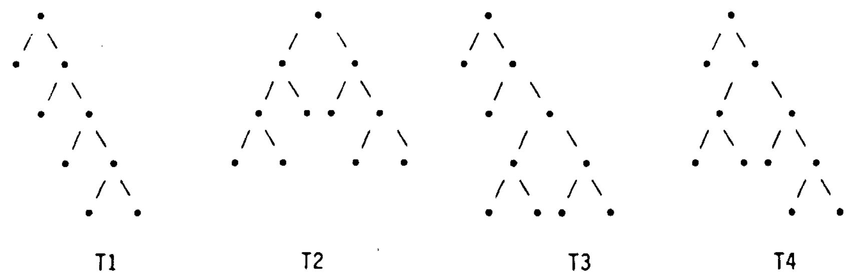


Figure 5. An abstract tree.



```
NL(T1) = 0
NL(T2) = (1 / 2) x [(2 - 1) x s(S1) + (2 - 2) x s(S2) + NL(S2)] = 1
NL(T3) = 1 / 8
NL(T4) = 1 / 4
```

Figure 6. Non-linearity measure for four trees.

defined as follows:

$$s(T) = 1 + \sum_{i=1}^{m} s(Si)$$

where $s(T) = 0$ if $T$ is a class value. It is assumed that $Si$ are sorted in increasing order of $s(Si)$. Non-linearities of four trees are shown in Figure 6.

*T*1 is absolutely linear; thus its non-linearity measure is zero. *T*2 is very close to being a balanced tree: non-linearity one. *T*3 is preferred to *T*4, i.e. this function is sensitive to the location of non-linearity within a tree (the lower a non-linearity occurs in a tree the lower (better) its measure).

## 4. EFFICIENCY MEASURE

Consider the example set of Figure 2. Two equally linear decision-trees for classifying this example set are shown in Figure 7. Labels on the arcs correspond to the number of examples per value of each attribute, Figure

```
       A1                                          A4
    4/    \7                                    4/    \7
    A2     A4                                   A2     A3
   / \   1/ \6                                 / \   2/ \5
 C1  C2  C1  A3                              C1  C2  C2  A2
           3/ \3                                       2/ \3
           C2  A2                                      C1  A1
             1/ \2                                       1/ \2
             C1  A5                                      C1  A5
                 / \                                         / \
               C2  C1                                      C2  C1

                   T1                                          T2
```

Figure 7. Trees with different execution cost.

```
                    A1
                   / \
                  t   f
                 /     \
t,t,f,t,t-cl            f,t,f,t,t-c2
t,f,t,f,f-cl            f,f,t,f,t-cl
t,f,f,t,f-cl            f,f,f,f,f-cl
t,t,f,t,f-c2            f,t,f,f,t-c2
t,f,t,t,f-c2
t,f,f,t,t-cl
t,t,t,t,f-c2
```

Figure 8. Attribute A1 divides the example set.

8 shows that attribute $A1$ divides the original example set into two example sets of size 7 and 4 each. Let $c(a_i)$ represent the execution cost of an attribute. There are 11 examples in the original example set and the execution cost for each tree can be computed on the basis of how early in the decision tree a classification takes place. One way of computing this cost is as follows:

Average cost for $T1 =$

$$[11 \times c(a_1) + 4 \times c(a_2) + 7 \times c(a_4) + 6 \times c(a_3) + 3 \times c(a_2) + 2 \times c(a_5)]/11$$

Average cost for $T2 =$

$$[11 \times c(a_4) + 4 \times c(a_2) + 7 \times c(a_3) + 5 \times c(a_2) + 3 \times c(a_1) + 2 \times c(a_5)]/11.$$

Note that the difference between $T1$ and $T2$ occurs at the second from the top. Using $T1$ the chance of an example being classified is only 1 out of 7 while using $T2$ it is 2 out of 7. Assuming the execution cost of each attribute has unit cost, $c(a_i) = 1$, the execution cost for trees $T1$ and $T2$ are 3.0 and 2.9 respectively. That is, $T2$ is about 3% more efficient than $T1$. The difference can be larger, see Figure 10 for more examples. Thus, attribute selection can have an effect on the average execution cost of a decision-tree. Clearly, it is desirable for attributes with high information content (entropy) to appear as early as possible in a decision-tree. This increases the probability that a classification will occur as soon as possible. Thus RG employs entropy as the selection criterion for increasing efficiency. The selection criterion must then consider the entropy. The entropy or information content of an attribute can be computed from the following formula [6]. An attribute's entropy is given by $M(C)$ minus $B(C, A)$ where $C$ is the example set and

$$M(C) = \sum_{i=1}^{n} - P_i \times \log_2 P_i$$

$$B(C, A) = (\text{probability that value for } A \text{ is } A_i) \times M(C_i)$$

where $n$ is the number of classes, $P_i$ is the occurrence probability of the

$i$th class, $A$ is an attribute, $A_i$ is a value for the attribute $A$ and $C_i$ is the example set after it has been split by $A$.

Note that the probabilities can be estimated from the relative frequencies in $C_i$ given that $C$ is a representative of the universe. For example, in order to compute the entropy of attribute $A1$ in Figure 2 we first must compute $B(C, A1)$. This computation requires the example set to be divided according to the values of $A1$ as in Figure 8.

The information content of the true and false branches of $A1$ can be computed as follows:

$$M(C_1) = -\tfrac{4}{7} + \log_2\tfrac{4}{7} - \tfrac{3}{7} \times \log_2\tfrac{3}{7} = 0.98522$$

$$M(C_2) = -\tfrac{2}{4} \times \log_2\tfrac{2}{4} - \tfrac{2}{4} \times \log_2\tfrac{2}{4} = 1.0.$$

The expected information content, $B(C, A1)$ of $A1$ is:

$$B(C, A1) = \tfrac{7}{11} \times 0{\cdot}98522 + \tfrac{4}{11} \times 1.0 = 0.99059.$$

The information content of $A1$ is $M(C)$, where $C$ is the original example set, minus $B(C, A1)$ or $0.99395 - 0.99059 = 0.0036$. The entropy for other attributes can be computed in this fashion. A selection based on this criterion will minimize the expected execution cost of decision-trees.

Entropy has been used without the linearity measure in ID3. It should not be surprising that decision trees constructed in such a fashion tend to be balanced rather than linear. For example, the ID3 solution for the example set in Figure 2 is shown in Figure 9, which also shows the tree generated by our algorithm.

Clearly, ID3 produces more efficient decision trees than RG. However, since we choose to sacrifice efficiency for linearity, decision-trees produced by RG are more desirable. If the added tree-synthesis cost can be accepted, then optimal efficiency (among equally linear candidate trees) can be guaranteed by substituting 'best-first with backtrack' for the 'best-first no backtrack' strategy borrowed from ID3.

```
       A4                                                    A2
    f/    \t                                              t/    \f
    A2      A3                                            A1      A3
  f/ \t   t/ \f                                         t/ \f   f/ \t
 C1   C2  C2   A2                                      A5  C2   C1   A4
             f/ \t                                   t/ \f          f/ \t
             C1   A1                                C1   C2         C1   C2
                f/ \t
                C1   A5
                   f/ \t
                   C2   C1

               RG                                   As solved by ID3
```

y measures.

## 5. OUTLINE OF RG

We adapted Bratko's measure of non-linearity and used an attribute selection criterion that promotes execution efficiency of the resulting decision-tree. RG incorporates the notions of linearity and efficiency into an AO* [13] search technique.

The state space for finding a decision tree is finite and decreasing with the number of variables since the number of attributes and examples are finite. An 'And/Or' tree is used to represent the state space. 'Or' nodes correspond to candidate attributes and 'And' nodes are subproblems that must be solved. The root can be considered as an 'And' node. Each node may be labelled as solved, closed or open. Solved nodes mean that a solution has been reached from this node. A closed node means that a solution under current consideration incorporates this node internally. A node is open if it is neither closed nor solved.

During the expansion of the search tree, an optimistic estimate for non-linearity is used in conformity with the AO* algorithm for searching 'And/Or' graphs. This estimate differentiates between total, partial and non-decider attributes. Thus, if there are total deciders among the candidate attributes, the search tree is expanded using them and the nodes are labelled as solved. All partial decider attributes are considered if no total deciders exist, and non-decider attributes are considered only if there are no total or partial deciders.

The optimal solution path is marked in the search tree according to: (i) non-linearity of the partially constructed decision tree; (ii) number of expected internal modes; (iii) the attribute's entropy measure. The entropy measure is used simply as a tie breaker between attributes which produce equally linear decision-trees. Thus, optimality with respect to linearity is guaranteed while efficiency is only enhanced. When RG terminates, the optimal decision-tree can be constructed by tracing markers from the root node to the bottom and recording the attributes and their values.

## 6. RESULTS WITH RG

RG was used to induce rules for some examples selected from the planning domain (construction of an arch and sorting a stack of blocks) and chess end-games (some examples from Shapiro's Ph.D. thesis [7]) in addition to some artificially constructed example sets. For the most part the rules synthesized by RG were more linear than those induced by ID3. The exceptions occurred when ID3 happened to construct a fully linear decision-tree. ID3 produces more efficient decision-trees than AOCDL or RG. This is to be expected because RG (and AOCDL) emphasizes the linearity criterion before efficiency. However, the decision-trees generated by RG were more efficient than those produced by AOCDL since this

| | RG | | AOCDL | | ID3 | |
|---|---|---|---|---|---|---|
| | NL | Cost | NL | Cost | NL | Cost |
| EX1 | 0 | 2.7 | 0 | 2.7 | 0 | 2.7 |
| EX2 | 0.5 | 2.91 | 0.5 | 3.0 | 1.0 | 2.54 |
| EX3 | 0 | 2.25 | 0 | 2.25 | 0.5 | 2.0 |
| EX4 | 1.0 | 2.76 | 1.0 | 3.5 | 2.11 | 1.12 |
| EX5 | 0 | 3.0 | 0 | 3.42 | 0 | 3.0 |

Figure 10. Performance analysis of programs.

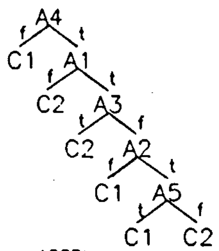
```
problem(ex1).

/* Attributes */
att(a1, (t.f.nil)).
att(a2, (f.t.nil)).
att(a3, (f.t.nil)).
att(a4, (f.t.nil)).
att(a5, (f.t.nil)).

/* Classes */
class(c1).
class(c2).

/* Examples */
ex(1,  (t.t.f.t.t.nil), c1).
ex(2,  (t.f.t.f.f.nil), c1).
ex(3,  (t.f.f.t.f.nil), c1).
ex(4,  (f.t.f.t.t.nil), c2).
ex(5,  (t.t.f.t.f.nil), c2).
ex(6,  (t.f.t.t.f.nil), c2).
ex(7,  (f.f.t.f.t.nil), c1).
ex(8,  (t.f.f.t.t.nil), c1).
ex(9,  (t.t.t.t.f.nil), c2).
ex(10, (f.f.f.f.f.nil), c1).
```

Figure 11. Input file for EX1; this can be explained by a linear decision-tree.



AOCDL,
RG,
ID3

Figure 12. Synthesized decision-trees for EX1.

```
problem(ex2).
/* Attributes */
att(a1, (t.f.nil)).
att(a2, (f.t.nil)).
att(a3, (f.t.nil)).
att(a4, (f.t.nil)).
att(a5, (f.t.nil)).
/* Classes */
class(c1).
class(c2).
/* Examples */
ex(1,  (t.t.f.t.t.nil), c1).
ex(2,  (t.f.t.f.f.nil), c1).
ex(3,  (t.f.f.t.f.nil), c1).
ex(4,  (f.t.f.t.t.nil), c2).
ex(5,  (t.t.f.t.f.nil), c2).
ex(6,  (t.f.t.t.f.nil), c2).
ex(7,  (f.f.t.f.t.nil), c1).
ex(8,  (t.f.f.t.t.nil), c1).
ex(9,  (t.t.t.t.f.nil), c2).
ex(10, (f.f.f.f.f.nil), c1).
ex(11, (f.t.f.f.t.nil), c2).
```

Figure 13. Input file for EX2; this cannot be explained by a linear decision-tree.

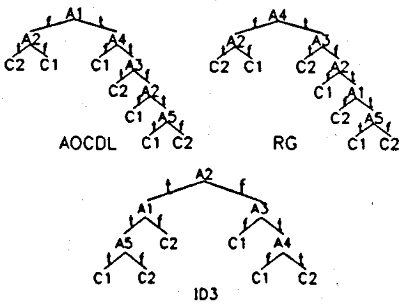


Figure 14. Synthesized decision-tree for EX2.

```
problem(ex3).

/* Attributes */

att(a1, (t.f.nil)).
att(a2, (f.t.nil)).
att(a3, (f.t.nil)).

/* Classes */

class(c1).
class(c2).
class(c3).
class(c4).

/* Examples */

ex(1,  (t.t.f.nil), c1).
ex(2,  (f.t.t.nil), c2).
ex(3,  (f.f.t.nil), c3).
ex(4,  (f.f.f.nil), c4).
```

Figure 15. Input file for EX3; a linear decision-tree but ID3 cannot find it.

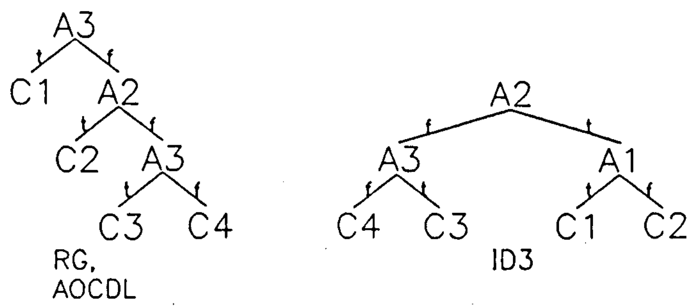


Figure 16. Synthesized decision-trees for EX3.

program, AOCDL, only optimizes the linearity criterion. The decision-trees generated by RG are more understandable than those generated by ID3, since RG promotes efficiency without destroying linearity. For an example see Figure 1. Five examples that demonstrate the differences between these programs are given in Figure 10. Complete listings of the examples may be obtained from the authors. Outline listings of the examples,

```
problem(building_tower).

/* Attributes */
att(aon, (f.b.c.nil)).
att(bon, (f.a.c.nil)).
att(con, (f.a.b.nil)).
att(cleara, (y.n.nil)).
att(clearb, (y.n.nil)).
att(clearc, (y.n.nil)).

/* Classes */

class(stop).
class(btoc).
class(ctof).
class(atob).
class(btof).
class(atof).

/* Examples */

ex(1 ,  (f.f.f.y.y.y.nil), btoc).
ex(2 ,  (f.f.a.n.y.y.nil), ctof).
ex(3 ,  (f.f.b.y.n.y.nil), ctof).
ex(4 ,  (f.a.f.n.y.y.nil), btoc).
ex(5 ,  (f.a.b.n.n.y.nil), ctof).
ex(6 ,  (f.c.f.y.y.n.nil), atob).
ex(7 ,  (b.f.f.y.n.y.nil), atof).
ex(8 ,  (b.f.a.n.n.y.nil), ctof).
ex(9,   (b.c.f.y.n.n.nil), stop).
ex(10,  (f.c.a.n.y.n.nil), btof).
ex(11,  (c.f.f.y.y.n.nil), atof).
ex(12,  (c.f.b.y.n.n.nil), atof).
ex(13,  (c.a.f.n.y.n.nil), btof).
```

Figure 17. Input file for EX4; building a tower in the blocks world.

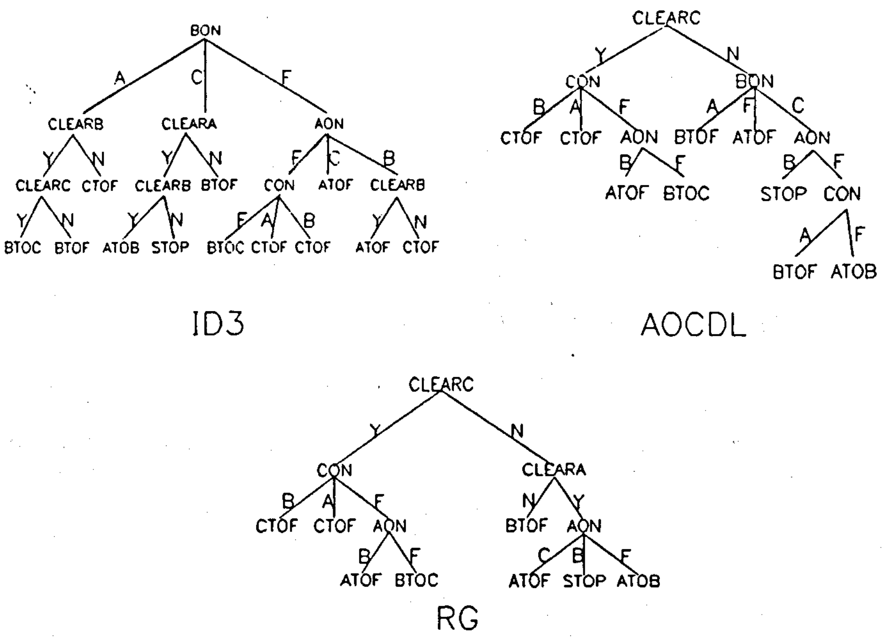


Figure 18. Synthesized decision-trees for EX4.

together with the trees generated by the different algorithms, are shown in Figures 11–20.

The table of Figure 10 and the more detailed listings of Figures 11–20 indicate that RG produces decision-trees that are as linear as those produced by AOCDL but more efficient. Also, decision-trees produced are more linear than those produced by ID3. Thus, RG has been successful in

```
problem(btoqs).

class(true).
class(false).

att(skrxp, f.t.nil). /*Can the BR achieve a skewer or BK attack the WP*/
att(bkona, f.t.nil). /*Is the BK on rank A in a position to aid the BR*/
att(bkon8, f.t.nil). /*Is the BK on file 8 in a position to aid the BR*/
att(bknwy, f.t.nil). /* Is the BK in the BR's way */
att(wkovl, f.t.nil). /* Is the WK overloaded */

ex(1, t.f.f.f.f.nil, true).
ex(2, f.t.f.f.f.nil, true).
ex(3, f.f.t.f.f.nil, true).
ex(4, f.f.f.t.f.nil, false).
ex(5, f.f.f.f.t.nil, true).
ex(6, f.f.f.f.f.nil, false).
ex(7, f.f.f.t.t.nil, false).
```

Figure 19. Input file for EX5; an example from ref. [7].

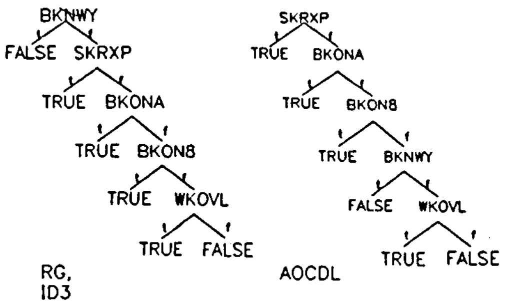


Figure 20. Synthesized decision-trees for EX5.

its goal, i.e. producing the most linear decision-tree while enhancing execution efficiency.

## 7. SUMMARY

An algorithm, RG, for producing human-understandable yet efficient decision-trees has been described and implemented. RG was implemented in PROLOG and tested using sample problems from Bratko [1] and Shapiro [7]. In all cases of difference, the decision-trees produced were more linear than decision-trees synthesized by ID3 and more efficient that decision-trees generated by AOCDL.

This algorithm is heuristically guided by linearity and efficiency measures resulting in the generation of more understandable decision trees. RG, combined with structured induction promises rapid construction of expert systems by permitting the expert to communicate his knowledge and experience through a simple yet flexible language.

**Acknowledgements**

We would like to thank Rina Dechter, Jim Moore, and Gary Silverman for providing many helpful suggestions throughout the project. Thanks are also due to the IBM Los Angeles Scientific Center for providing resources and support.

## REFERENCES

1. Bratko, I. (1983) Generating human-understandable decision rules, Working paper, E. Kardelj University, Liubljana, Yugoslavia.
2. Michie, D. (1982) The state of the art in machine learning. In *Introductory readings in expert systems* (ed. D. Michie) pp. 208–228. Gordon & Breach, London.
3. Vere, S. A. (1978) Inductive learning of relational productions. In *Pattern-directed inference systems* (eds D. A. Waterman and F. Hayes-Roth) pp. 281–95. Academic Press, New York.
4. Michalski, R. S. (1980) Pattern recognition as rule-guided inductive inference, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-2,** 349–361.
5. Hayes-Roth, F. and McDermott, J. (1977) Knowledge acquisition from structural description. *Proc. IJCAI-5,* pp. 356–362.

6. Quinlan, J. R. (1983) Learning efficient classification procedures and their applications to chess end-games In *Machine learning: an artificial intelligence approach* (eds R. S. Michalski, J. Carbonell, and T. Mitchell) pp. 463–82. Tioga, Palo Alto, Calif.
7. Shapiro, A. (1987) *Structured induction in expert systems.* Addison Wesley, Wokingham, England, and New York.
8. Martelli, A. and Montanari, U. (1973) Optimizing decision trees through heuristically guided search, *Commun. ACM* **21,** 1025.
9. Michie, D. (1981) 'Mind-like' capabilities in computers: a note on computer induction. *Cognition* **12,** 97–108.
10. Shapiro, A. and Niblett, T. (1982) Automatic induction of classification rules for a chess endgame. In *Advances in computer chess 3* (ed M. R. B. Clarke) pp. 73–92. Pergamon Press, Oxford.
11. Dechter, R. and Michie, D. (1984) Structured induction of plans and programs, IBM Los Angeles Scientific Center report.
12. McLaren, R. (1983) *Expert-Ease user manual.* Intelligent Terminals Ltd., Glasgow.
13. Nilsson, N. J. (1980) *Principles of artificial intelligence.* Tioga, Palo Alto, Calif.
14. Sammut, C. and Banerji, R. B. (1981) Learning concepts by asking questions, Working paper, Department of Computer Science, University of Illinois at Urbana-Champaign.
15. Shapiro, E. Y. (1981) *Inductive inference of theories from facts.* Department of Computer Science, Yale University.

## FURTHER READING

Arbab, B. and Michie, D. (1985) Generating rules from examples. *Proc. IJCAI-9.*

Attneave, F. (1959) Applications of information theory to psychology: A summary of basic concepts, methods and results. University of Oregon.

Paterson, A. and Niblett, T. (1982) *ACLS user manual.* Intelligent Terminals Ltd., Edinburgh.

Quinlan, J. R. (1982) Semi-autonomous acquisition of pattern-based knowledge. In *Machine Intelligence 10* (eds J. E. Hayes, D. Michie, and Y.-H. Pao). Ellis Horwood, Chichester.

Roberts, G. M. (1977) M.S. thesis, An implementation of PROLOG. University of Waterloo, Department of Computer Science.

Roberts, G. M. (1983) PROLOG *user's manual, version 1.4.* University of Waterloo, Department of Computer Science.

13

# Decision Trees and Multi-Valued Attributes

J. R. Quinlan
School of Computing Sciences,
New South Wales Institute of Technology, Australia

**Abstract**
Common induction systems that construct decision-trees have been reported to operate unsatisfactorily when there are attributes with varying numbers of discrete possible values. This paper highlights the deficiency in the evaluation of the relevance of attributes and examines a proposed solution. An alternative method of selecting an attribute is introduced which permits the use of redundant attributes. Results of experiments on two tasks using the various selection criteria are reported.

## 1. INTRODUCTION

As knowledge-based expert systems play an increasingly important role in artificial intelligence, more attention is being paid to the problem of acquiring the knowledge needed to build them. The traditional approach involving protracted interaction between a knowledge engineer and a domain expert is viable only to the extent that both these resources are available; this approach will not meet the apparently exponential growth in demand for expert systems. A solution to this dilemma requires rethinking the way knowledge-based products are built. An example of this reappraisal of methodology appears in Michie (1983), and is based on the principle of formalizing and refining the knowledge implicit in collections of examples or data bases.

Dietterich and Michalski (1983) give an overview of methods for learning from examples. There are many such, all based on the idea of inductive generalization. One of the simplest of these methods dates back to work by Hunt in the late fifties (Hunt *et al.*, 1966). Each given example, described by measuring certain fixed properties, belongs to a known class and the 'learning' takes the form of developing a classification rule that can then be applied to new objects. Simple though it may be, derivatives of this method have achieved useful results; Kononenko *et al.* (1984), for example, have managed to generate five medical diagnosis systems with minimal reference to diagnosticians.

In the course of their work, Kononenko *et al.* uncovered a deficiency in the basic machinery being used, and this paper focuses on that shortcoming. We first formalize the inductive task and present analytical support for the existence of a deficiency. The solution proposed by Kononenko *et al.* is discussed and an alternative method introduced. A series of trials of the various methods on two classification tasks is then reported, revealing their relative merits.

## 2. CONSTRUCTING DECISION-TREES

We imagine a universe of *objects*, each described in terms of a fixed number of *attributes* or properties. Each attribute has its own (small) set of discrete attribute values. Each object belongs to one of several mutually exclusive classes. For example, we could have the following scenario:

| | |
|---|---|
| objects: | people |
| attributes: | colour of hair (with attribute values red, brown, fair, grey, black) |
| | colour of eyes (brown, black, blue) |
| | height (tall, medium, short) |
| classes: | unremarkable appearance, unusual appearance. |

The concept learning task of interest here can now be stated briefly as:

given: a collection of objects (and their descriptions) whose class membership is known

find: a classification rule, couched in terms of the attributes, that will assign any object to its class.

The given set of objects is usually referred to as the *training set.* The method used to find a rule is induction in which observations about the training set are generalized so as to apply to other, as yet unseen objects. While it is possible to ensure that the developed rule works for all members of the training set, the correct performance of the rule on other objects cannot usually be guaranteed. Instead, we rely on heuristic guides such as Occam's Razor: among all rules that accurately account for the training set, the simplest is likely to have the highest success rate when used to classify objects that were not in the training set.

In this discussion we will assume that there are only two classes, *Y* and *N,* although all the results can be extended to an arbitrary number of classes in a straightforward way. We will also limit ourselves to classification rules expressed as decision trees. Each interior node of such a tree is a test, based on a single attribute, with a branch for each possible outcome of the test. Each leaf of a decision-tree is labelled with a class. An object is classified by starting at the root of the tree, performing the test, taking the branch appropriate to its outcome, and continuing the subtree at that branch. When a leaf is eventually encountered, the object is taken to be a member of the class associated with the leaf.

Forming a decision-tree by induction from a training set comes down to deciding, from the descriptions and known classes of objects in the training set, which attribute-based test to use for the root of the decision tree. For this test (with $v$ outcomes, say) will partition the training set into $v$ blocks, one for each branch emanating from this root node. Each block can be treated as a training set in its own right and the same procedure applied recursively until all (sub) training sets contain objects of a single class. A method of choosing a test to form the root of the tree will be referred to as a *selection criterion.*

One obvious choice for an attribute-based test is to branch on the value of an attribute, creating a separate path for each possible value it can have. Choosing a test for the root could be carried out by the trivial algorithm: choose the first attribute first, then the second attribute and so on. However, the decision-tree built by this procedure would not be expected to reflect any structure in the training set and so would have poor predictive performance. A better strategy, employed for example by ACLS (Michie, 1983; Shapiro, 1983; Shepherd, 1983) and ID3 (Quinlan 1982, 1983 a, b), is to use an information-theoretic criterion as follows. If the training set contains $y$ objects from class $Y$ and $n$ from class $N$, the information that needs to be supplied by a classification rule for the set can be related to the relative frequencies of class membership by the function

$$I(y, n) = -\frac{y}{y+n}\log_2\left(\frac{y}{y+n}\right) - \frac{n}{y+n}\log_2\left(\frac{n}{y+n}\right).$$

Now, let $A$ be an attribute with possible values $A_1, A_2, \ldots, A_v$, and let $y_i$ and $n_i$ denote the numbers of objects of class $Y$ and $N$ respectively that have the $i$th value $A_i$ of $A$. If attribute $A$ was chosen as the root of the decision-tree, with a branch for each of its $v$ possible values, the information that would need to be supplied by the (sub) tree corresponding the the branch for $A_i$ is similarly

$$I(y_i, n_i).$$

Weighting each branch of the decision-tree by the proportion of objects in the training set that belong to that branch, we can write the expected information requirement after testing attribute $A$ as

$$E(A) = \sum_{i=1}^{v} \frac{y_i + n_i}{y+n} I(y_i, n_i).$$

Naturally, the expected information needed after testing attribute $A$ is generally less than the information needed before any attribute is tested. The information gained by branching on attribute $A$ is just

$$\text{gain}(A) = I(y, n) - E(A).$$

The information-based criterion referred to earlier can be expressed simply as: choose the attribute whose information gain is maximal. In the following, this will be called the *original criterion.*

## 3. MULTI-VALUED ATTRIBUTES

Kononenko *et al.* (1984) have developed an inductive inference system ASSISTANT and used it to built classification rules in several medical domains. At one stage of its evolution, their system used the original criterion of information gain to select attributes as above. In the course of their experiments they encountered a problem when the attributes being compared had different numbers of values. In one study, medical specialists were of the opinion that the attribute 'age of patient', with nine discrete ranges, was being chosen over more relevant attributes with fewer values. The choice of an inappropriate attribute results in excessive fragmentation of the training set; structure in the set becomes harder to detect and the performance of the classification rule on unseen objects may be degraded. In this case, the opinion of the specialists was borne out by the fact that, when the attribute was omitted altogether, the induced classification rule gave better results.

Let us analyse the problem in more abstract terms. Suppose we form an attribute $A'$ which is identical to $A$ except that two of the attribute values, $A_1$ and $A_2$ say, are collapsed into a single value $A'_{1+2}$. $A'$ then has $v-1$ values $A'_{1+2}, A'_3, \ldots, A'_v$, where there are now $y_1+y_2$ and $n_1+n_2$ objects from classes $Y$ and $N$ respectively that have value $A'_{1+2}$ of the new attribute $A'$. Let us examine the difference in information gain between $A$ and $A'$. Since $I(y, n)$ is unchanged, this difference is

$$E(A') - E(A).$$

For values of $i > 2$, corresponding terms in this difference cancel so the difference reduces to terms related to $A_1$ and $A_2$ on one hand and $A'_{1+2}$ on the other. The difference can then be written as

$$\frac{y_1+y_2+n_1+n_2}{y+n} I(y_1+y_2, n_1+n_2) - \frac{y_1+n_1}{y+n} I(y_1, n_1) - \frac{y_2+n_2}{y+n} I(y_2, n_2).$$

The minimum value of this difference can be found by equating its partial derivative with respect to $y_1$ to zero. The minimum value occurs when

$$\frac{y_1}{y_1+n_1} = \frac{y_2}{y_2+n_2} = \frac{y_1+y_2}{y_1+n_1+y_2+n_2}$$

which gives the minimum value of the difference as zero. The upshot of this analysis is that the information gain attributable to $A$ will generally exceed that attributable to $A'$, the two gains only being equal if the proportions of class $Y$ and class $N$ objects in the two merged attribute values are identical.

Now let us look at the situation from the other side of the coin. Suppose that the values of attribute $A$ are sufficiently 'fine' for the classification task at hand. If we were arbitrarily to increase the number of values of $A$ by subdividing existing values, we would not expect to increase the usefulness of $A$ for a classification rule; on the contrary, we would intuitively expect the excessive fineness of $A$ to obscure structure that may exist in the training set. But the above analysis shows that the information gain of the new, finer $A$ will generally be increased, thereby boosting its chances of being chosen as the most relevant attribute. By analogy, there would seem to be a bias in the information gain criterion towards attributes with larger numbers of values. This analysis supports the empirical finding of Konenenko *et al.*

## 4. BINARY TESTS

The remedy implemented in ASSISTANT is the requirement that all tests have only two outcomes. If we have an attribute $A$ as before with $v$ values $A_1, A_2, \ldots, A_v$, the decision-tree no longer has a branch for each possible value. Instead, a subset $S$ of the values is chosen and the tree has two branches, one for all values in the set and one for the remainder. The information gained is then computed as if all values in $S$ were amalgamated into one single attribute value and all remaining values into another. In this selection criterion, referred to as the *subset criterion,* the test chosen for the root uses the attribute and subset of its values that maximizes the information gain. Kononenko *et al.* report this modification led to smaller (but less structured) decision-trees with an improved classification performance. In one medical domain, for example, the decision-tree formed from a training set of 1300 objects was reduced from 525 to 157 nodes, and its classification accuracy on 545 unseen objects improved from 62 per cent to 66 per cent.

Limiting decision trees to a binary format is reminiscent of the original concept learning system CLS (Hunt *et al.*, 1966). In that system, each test was of the form 'attribute $A$ has value $A_i$', with two branches corresponding to true and false. This is clearly a special case of the test implemented in ASSISTANT, which permits a set of values, rather than a single value, to be distinguished from the others. CLS, however, did not use an information-theoretic measure to evaluate tests, but rather employed a lookahead scheme based on a system of measurement and misclassification costs. Nevertheless, designating a single value and evaluating tests using information gain as before seems worthwhile exploring as a comparator for ASSISTANT's selection criterion, and will be referred to as the *single-value criterion.*

If all tests must be binary, there can be no bias in favour of attributes with large numbers of values and so the objective has certainly been achieved. It could be argued, however, that ASSISTANT's remedy has undesirable side-effects that have to be taken into account. First, it could lead to decision-trees that are even more unintelligible to human experts than is ordinarily the case, with unrelated attribute values being grouped together and multiple tests on the same attribute. More importantly, the modified procedure can require a large increase in computation. An attribute $A$ with $v$ values has $2^v$ value subsets and, when trivial and symmetric subsets are removed, there are still $2^{v-1}-1$ different ways of specifying the distinguished subset of attribute values. The information gain realized with each of these must be investigated, so a single attribute vith $v$ values has a computational requirement similar to $2^{v-1}-1$ binary attributes. This is not of particular consequence if $v$ is small, but the approach would appear infeasible for an attribute with 20 values. There are applications for which such a large number of attribute values is not unreasonable; for example, the attribute 'family' for Australian spiders would have 39 values (Clyne, 1969).

## 5. NORMALIZING THE GAIN

Another method of overcoming the problem posed by attributes with different numbers of values would be to normalize the information gain in some way. This was attempted by Kononenko *et al.* (1984): if an attribute had $v$ values, the normalized gain was computed as the 'raw' gain divided by $\log_2(v)$. The results achieved with this procedure were unsatisfactory, as very important attributes with large numbers of values were now discriminated against, at least near the root of the tree. For example, an attribute with eight values would have to achieve three times the information gain of a binary-valued attribute if it were to be the chosen attribute.

## 6. GAIN RATIO

This paper suggests an alternative information-based criterion that resembles a normalized gain, although the rationale for the criterion is quite different.

Consider again our training set containing $y$ and $n$ objects of class $Y$ and $N$ respectively. Let attribute $A$ have values $A_1, A_2, \ldots, A_v$ and let the numbers of objects with value $A_i$ of attribute $A$ be $y_i$ and $n_i$, respectively. Enquiring about the value of attribute $A$ itself gives rise to information, which can be expressed as

$$IV(A) = -\sum_{i=1}^{v} \frac{y_i + n_i}{y + n} \log_2\left(\frac{y_i + n_i}{y + n}\right).$$

Notice that this information measure is unrelated to the utility of $A$ for classification purposes. For example, if

$$y_1 = y_2 = \cdots = y_v; \quad \text{and}$$

$$n_1 = n_2 = \cdots = n_v$$

attribute $A$ would be useless as the root of the decision-tree, and yet the information from determining the value of attribute $A$ would be maximal.

$IV(A)$ thus measures the information content of the answer to the question, 'What is the value of attribute $A$?' As discussed earlier, gain($A$) measures the reduction in the information requirement for a classification rule if the decision tree uses attribute $A$ as root. Ideally, as much as possible of the information provided by determining the value of an attribute should be useful for classification purposes or, equivalently, as little as possible should be 'wasted'. A good choice of attribute would then be one for which the ratio

$$\text{gain}(A)/IV(A)$$

is as large as possible. This ratio, however, may not always be defined—$IV(A)$ may be zero—or it may tend to favour attributes for which $IV(A)$ is very small. We therefore propose the following criterion: from among those attributes with an average-or-better gain, select the attribute that maximizes the above ratio. This will be called the *ratio criterion.*

## 7. EMPIRICAL INVESTIGATION

The various criteria for selecting attributes as discussed in earlier sections were embodied in the straightforward tree-constructing procedure and evaluated on a family of tasks. This family was derived from an existing classification task, with a universe of 551 objects described in terms of 39 two-valued attributes for which the smallest known decision-tree contained 175 nodes (although smaller trees were discovered in the course of these experiments). In order to observe the effects of multi-valued attributes in stark relief, related tasks were synthesized by collapsing four of the attributes into a single attribute; these tasks thus had 36 attributes, one of them having 16 values and the remainder two values.

Three different choices of the four attributes to be combined into a single attribute were as follows:

$D1$: the two most important attributes were combined with two attributes of limited use

*D*2: the attributes were chosen to produce the most even distribution over the 16 values of the combined attribute

*D*3: the attributes were chosen to produce the most uneven distribution, subject to the requirement that all 16 values were represented.

Each selection criterion was evaluated on the original problem (D0) and on all the derived tasks. The same procedure was followed in each case. First, the entire 551 objects were presented as the training set to observe the size of the resulting decision-trees. Next, 20 randomly selected subsets containing 50 per cent of the 551 objects were set up and used as training sets. Since these training sets were incomplete, the decision-trees formed from them were not exact: each was tested on the remaining 50 per cent of the objects to measure the number of classification errors that resulted. Finally, to simulate forming more inaccurate classification rules, a similar procedure was followed using 20 per cent of the objects for the training set and evaluating the decision trees on the remaining 80 per cent.

The results of these experiments are summarized in Table 1 and Figures 1 and 2. Table 1 shows the sizes of the decision trees obtained from all 551 objects. For the original task (D0) in which all attributes are two-valued, the subset and single-value selection criteria are identical to the original, but noticeable differences emerge on the derived tasks. The ratio criterion does very well on the original task, giving a decision-tree of 143 nodes that is considerably smaller than any other known correct tree for this task. The same selection criterion, however, produces a much larger decision tree for task D1.

The most important characteristic of a good selection criterion, though, is that it should lead to decision-trees that accurately classify unseen objects. Figure 1 refers to the experiments in which decision-trees formed from half of the 551 objects were tested on the remaining half. For each task and selection criterion, the figure shows the 95 per cent confidence interval for the mean number of classification errors over the

Table 1. Number of nodes in correct decision tree.

| Selection criterion | D0 | D1 | D2 | D3 |
|---|---|---|---|---|
| Original | 175 | 205 | 187 | 187 |
| Subset | 175 | 205 | 169 | 163 |
| Single-value | 175 | 179 | 167 | 185 |
| Ratio | 143 | 265 | 179 | 179 |

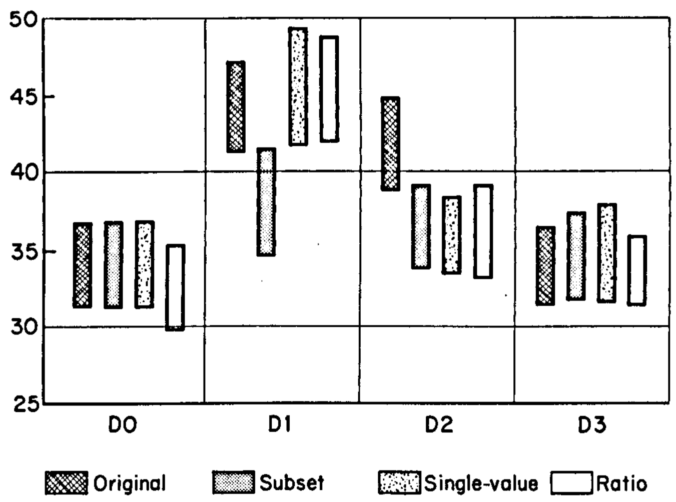


Figure 1. Mean number of errors with training-set of 275 objects.

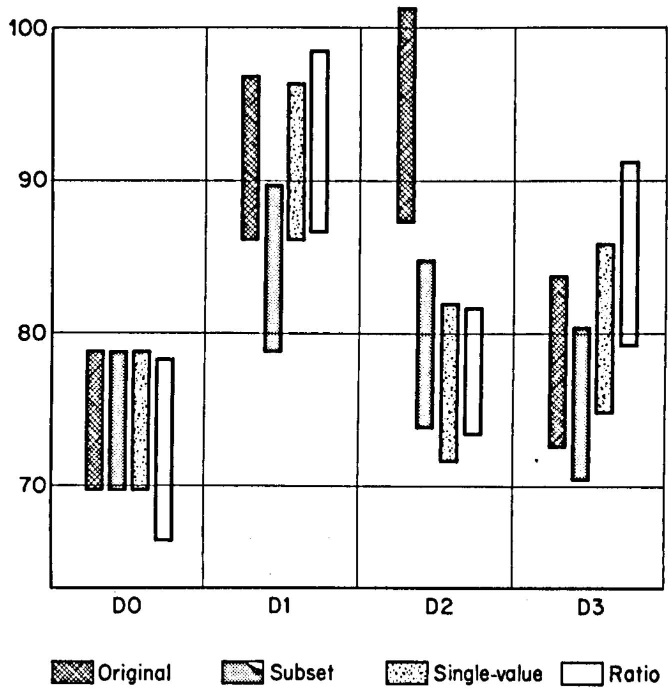


Figure 2. Mean number of errors with training set of 110 objects.

20 trials. These indicate that the subset criterion is significantly better on D1 while the original criterion is clearly worse on D2. Figure 2 refers to the similar experiments in which the training set contained 20 per cent of the objects and the resulting decision-trees were then tested on the 80 per cent of unseen objects. Once again a similar pattern emerges.

These results support the finding of Kononenko *et al.* that the original selection criterion can be somewhat deficient in a task with multi-valued attributes. For task D2 in Figure 2, changing from the original to the subset selection criterion improved the mean classification accuracy on unseen objects from 79 per cent to 82 per cent, and this difference would probably increase if more multi-valued attributes were involved.

## 8. REDUNDANT ATTRIBUTES

All selection criteria appear to have more difficulty with the task D1, as seen in both the size of decision-tree for the complete training set and the errors made by decision-trees constructed from partial training sets. Recall that this task aggregates both important and unimportant attributes, and thereby models a common real-world situation in which coarse values of an attribute are all that is required for classifying most objects, but much more precise values are needed for rarer cases. Two examples should illustrate the idea. In a thyroid diagnosis system (Horn *et al.*, 1985) many cases can be classified by knowing simply whether the level of each measured hormone is normal or not, but some cases require the level to be divided into as many as seven subranges. The study on Australian spiders mentioned earlier divides the 39 families into six groups, where the group alone often provides sufficient information for classification purposes.

The obvious remedy is to incorporate redundant attributes, each measuring the same property at different levels of precision appropriate to different classification needs. In the examples above, we might have both hormone level (seven values) and whether normal (two values), and both spider family (39 values) and group (six values). It would seem that the human experts, who provide the attributes in the first place, would have little difficulty in specifying these different precision levels useful for particular subsets of objects.

Let us now see what effect the introduction of a redundant attribute might be expected to have on the decision trees produced by the various selection criteria. Suppose $A$ is some attribute with a full complement of values and $A'$ is a redundant attribute with a lower level of precision, i.e. at least one value of $A'$ corresponds to a subset of values of $A$. We have shown earlier that the information gain using $A'$ can never exceed that using $A$, so the original selection criterion will never prefer $A'$ to $A$. That is, adding the redundant attribute $A'$ will have no effect on the decision-tree formed. When the subset selection criterion is used, it is apparent that any subset of the values of $A'$ can also be expressed as a subset of the more finely divided values of $A$, so including redundant attribute $A'$ will not increase the range of tests available. However, some value of $A$ may not be represented in a small training set while the corresponding coarser value of $A'$ is represented, so tests derived from small sets of objects may be more accurate using $A'$ rather than $A$. In the case of the single-value criterion, however, adding $A'$ may have a beneficial effect by broadening the range of possible tests, as one attribute value of $A'$ may correspond to a subset of the values of $A$. Finally, the attribute information $IV(A')$, will generally be less than $IV(A)$, so the introduction of $A'$ would be expected to have an effect on the ratio criterion, although whether this effect is beneficial or otherwise is not clear. To summarize: the addition of a redundant attribute will not change the decision-trees produced by the original criterion, should not have a dramatic effect on those produced by the subset criterion, but may alter the trees produced using the single-value and ratio criteria.

These observations were tested by rerunning the previous trials, this time including both the original binary attributes as well as the composite 16-valued attributes. Each original attribute is then redundant in the

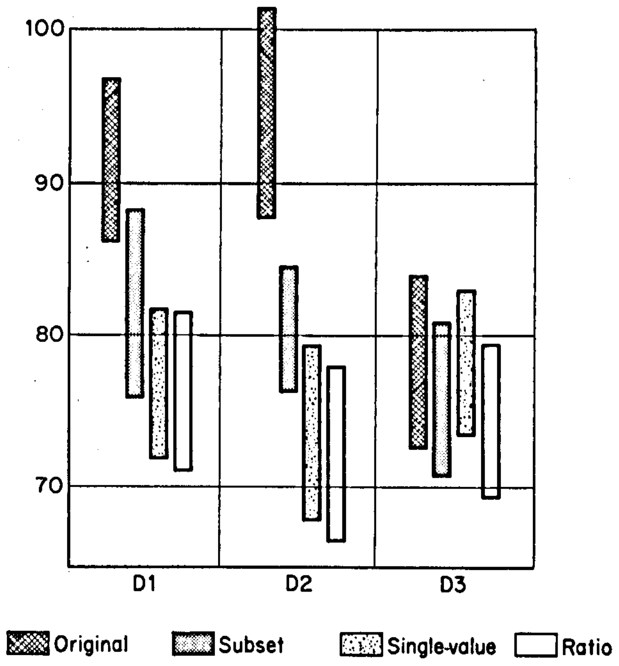


Figure 3. Mean number of errors, original attributes included, with training-set of 110 objects.

sense above, because each of its values corresponds to eight values of the composite attribute. Each of the tasks D1, D2 and D3 now has 40 attributes, one with 16 values and the rest with two. As before, the trials included 20 'runs' on each task, selecting a fixed proportion of the set of 551 objects as a training set and testing the tree produced on the remainder.

Figure 3 summarizes the results when 110 objects (20 per cent) were used as a training set, and shows the mean number of errors when the trees were used to classify each of the other 80 per cent of the objects. If this figure is compared to the corresponding sections of Figure 2, the following points emerge. As expected, the inclusion of the additional redundant attributes has no effect on the trees produced using the original selection criterion. There are small changes in the mean error with the subset criterion, and significant improvements with the single-value and ratio criteria. There is a particularly noticeable decrease in mean errors with the ratio criterion on D1 and D3. Notice also that the ratio criterion now gives marginally lower errors on all tasks than the other criteria.

## 9. CONFIRMING EXPERIMENT

The results above were put to the test on a completely unrelated classification task. In this task there were 1987 objects described in terms of 14 attributes, five with three values and the remaining nine with two. Despite the larger number of objects, this is a much simpler classification task as a correct decision tree with only 48 nodes was previously known.

Twenty training sets were prepared by randomly selecting half of the 1987 objects. For each of the four selection criteria, decision-trees were formed from each of these training sets and tested on the remaining objects. As would be expected from the simpler concept being formed and from the larger training sets, there were relatively few errors when the trees were used to classify the unseen objects. It was observed that, for three of the multi-valued attributes, one attribute value was more important than the others. Following that philosophy of redundant attributes discussed above, a redundant binary attribute was added for each of these three. In this redundant attribute, the two less important values were merged into a single value. The runs were repeated, this time using the augmented set of 17 attributes.

The results are summarized in Figure 4. Notice that, since no attribute has more than three values, selecting a non-trivial subset of values is equivalent to selecting a single value, so the subset and single-value criteria give identical results. In the first runs, the original and ratio criteria emerge as less useful than the others, because the decision-trees formed using them give a higher error rate. When the three redundant

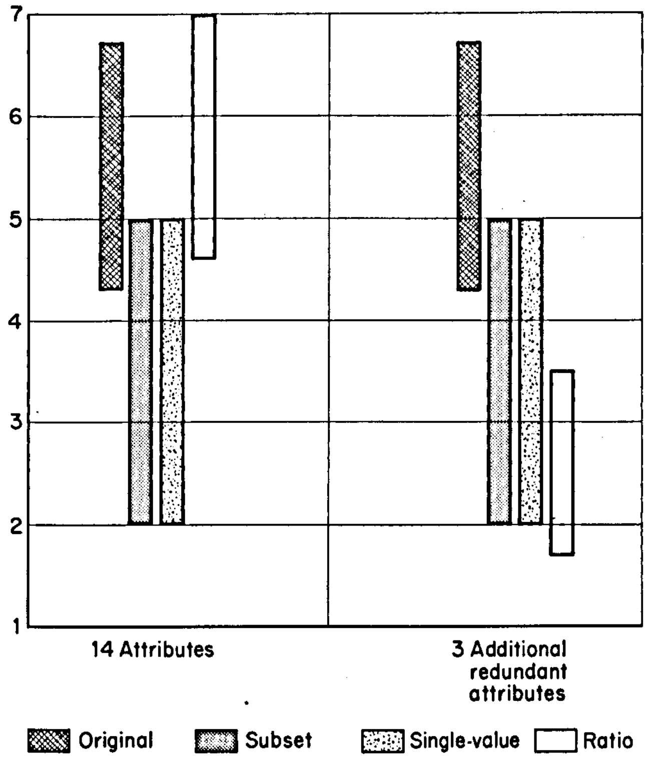


Figure 4. Mean number of errors with training-set of 993 objects.

attributes are added, however, only the ratio criterion is affected: it now gives significantly better results than any of the other three criteria.

## 10. CONCLUSION

Several observations can be made regarding these results. Analysis and experiments both support the findings of Kononenko *et al.* (1984) regarding the deficiency of the original selection criterion when attributes with differing numbers of values are present. This deficiency will tend to favour attributes with larger numbers of values, producing a decision-tree that has a higher error rate when classifying unseen objects.

The solution proposed by Kononenko *et al.* is to restrict tests in decision trees to binary outcomes, i.e. whether or not the value of an attribute is in a designated set. This has been found to reduce the size and improve the accuracy of decision-trees. However, the computational requirements of the subset selection criterion may make it infeasible for tasks containing attributes with many values. This technique has been compared to a similar binary restriction explored by Hunt *et al.* (1966), the single-value criterion, which makes no such exponential computational demands. The single-value criterion has also been found to generate slightly more accurate decision-trees than the original criterion.

We have also proposed and investigated a selection criterion based on the ratio of information gain to attribute information. This has been found generally to perform about as well as the single-valued criterion. However, it has two noteworthy advantages. It does not restrict the decision-tree to a binary format which may be awkward and unnatural for some applications. More importantly, it is able to benefit from the provision of redundant attributes whose levels of detail, as expressed by their number of possible values, can be chosen to suit different classification needs. When suitable redundant attributes are provided, the ratio criterion has been observed to outperform the other three criteria, even though its computational requirements are roughly equivalent to those of the original selection criterion.

## REFERENCES

Clyne, D. (1969) *A guide to Australian spiders.* Nelson, Sydney.

Dietterich, T. G. and Michalski, R. S. (1983) A comparative review of selected methods for learning from examples. In *Machine learning* (eds R. S. Michalski, J. Carbonell and T. Mitchell). Tioga, Palo Alto, Calif.

Horn, K., Compton, P., Lazarus, L., and Quinlan, J. R. (1985) The implementation of an expert system for the interpretation of thyroid assays in a clinical laboratory. *Australian Computer Journal* **17**, 1.

Hunt, E. B., Marin, J., and Stone, P. (1966) *Experiments in induction.* Academic Press, New York.

Kononenko, I., Bratko, I., and Roskar, E. (1984) Experiments in automatic learning of medical diagnostic rules, *Technical report,* Jozef Stefan Institute, Ljubljana, Yugoslavia.

Michie, D. (1983) Inductive rule generation in the context of the Fifth Generation, *Proc. Int. Machine Learning Workshop,* University of Illinois at Urbana-Champaign.

Quinlan, J. R. (1982) Semi-autonomous acquisition of pattern-based knowledge. In *Machine intelligence 10* (eds J. E. Hayes, D. Michie, and Y.-H. Pao). Ellis Horwood, Chichester.

Quinlan, J. R. (1983a) Learning efficient classification procedures. In *Machine learning: an artificial intelligence approach* (eds R. S. Michalski, J. Carbonell, and T. Mitchell). Tioga, Palo Alto, Calif.

Quinlan, J. R. (1983b) Learning from noisy data. *Proc. Int. Machine Learning Workshop,* University of Illinois at Urbana-Champaign.

Shapiro, A. (1983) The role of inductive learning in expert systems. Ph.D Thesis, University of Edinburgh.

Shepherd, B. A. (1983) An appraisal of a decision tree approach to image classification. *Proc. IJCAI-8* (Karlsruhe).

# 14
# RuleFactory: A New Inductive Learning Shell

S. Renner
Department of Computer Science,
University of Illinois at Urbana-Champaign, USA

## 1. INTRODUCTION

RuleFactory, an inductive learning program under development at the University of Illinois, is used to construct classification rules from expert-supplied examples. In this regard it is similar to the existing program ACLS and Expert-Ease based on ID3 [1, 2]. RuleFactory uses the induction algorithm contained in these two programs and consequently produces rules in the same 'decision-tree' format. However, several new features have been added to RuleFactory, resulting in a more powerful and useful tool.

RuleFactory can be thought of as a computerized apprentice, learning one or more classification rules for a problem domain. The user's role is that of a teacher, instructing RuleFactory by supplying pre-classified relevant examples from the domain. The user is also responsible for identifying the important features of the domain, called *attributes.* Each example supplied by the user is expressed as a list of attribute values together with the correct class value; e.g. one such example might say: 'When it is raining, and you are outside, and you are not already wet, you should use an umbrella.' From these examples, RuleFactory produces a rule which determines the class value of every instance in the domain. This rule can be inspected and tested by the user, and corrected by adding new examples or new attributes to the problem.

One new feature in RuleFactory is a facility which supports *structured induction,* a technique originated by Niblett and Shapiro [3] and further developed by Shapiro [4] to produce comprehensible rules for complex domains. In structured induction, the user repeatedly divides a problem into subproblems, treating each attribute as a separate problem for which he supplies new attributes and new examples. This process continues until a group of simple, *primitive* attributes emerges. RuleFactory allows the user to turn any attribute into a problem in this fashion, keeping track of the relations between attributes that result. Also, if the value of a non-primitive attribute is required during rule testing, RuleFactory obtains it by calling the rule for this attribute as a subprocedure, instead of asking the user for it.

RuleFactory provides a facility for explaining the decisions made by its classification rules. This facility is based on a technique developed by Shapiro and Michie [5] called *self-commenting,* subsequently refined on the basis of proposals from I. Bratko (personal communication). Self-commenting involves attaching a piece of text to each attribute and attribute value. By printing the attribute text when a rule is entered and the value text when the rule is finished, RuleFactory can produce a complete explanation of how the final result was obtained.

It is frequently possible to write external routines which compute the values of primitive attributes. RuleFactory provides a mechanism for communicating with these external functions, allowing RuleFactory to obtain attribute values from them instead of from the user. RuleFactory also generates C-coded versions of its decision rules. This code can be compiled together with the user's code to form a system which can execute the classification rules independently of RuleFactory, on any system which supports the standard C input/output library.

RuleFactory is a highly screen-oriented program. The terminal screen is divided into several different windows, each displaying some aspect of the current problem domain. Most of the user's input is entered as single-keystroke commands, which either modify the data displayed in the windows on the screen, or change the type and contents of these windows. A menu for the single-key commands is continually displayed, and a complete list of the longer commands can be obtained at any time.

A prototype version of RuleFactory has been implemented on a Vax 11/780 running 4.2 bsd UNIX.

## 2. DEFINITIONS

A RuleFactory *attribute* can be thought of as a function which classifies the members of the domain according to some important feature. For example, the number of high-card points in a hand is one attribute in the game of contract bridge; the domain of the corresponding function is the set of all bridge hands, and the range is an integer from 0 to 40. The range of a RuleFactory attribute must be a scalar or a single number; the domain can be whatever the user likes.

The purpose of RuleFactory is to create a *rule* which defines the classification function for an attribute in terms of other, simpler attributes. RuleFactory expresses such a rule in the form of a decision-tree. The internal nodes in a decision-tree represent a lower-level attribute to be tested. Each external node contains one of the possible outcomes of the rule. The rule is evaluated by starting at the root node, and at each test node using the test result to select one branch until an external node is reached; the result of the rule is the value of this final node.

**Problem domain:** KP(a7)KR
Each instance in the domain is a chess endgame position.
White has king and pawn, Black has king and rook.
The pawn is on square a7.

**Class attribute:** pa7
With White to move, is the position won for White?
The range of this attribute is:
Won—the position is won for White.
Not—the position is drawn, or won for Black.

**Attribute list:**

| | |
|---|---|
| rimmx— | Can the Black rook be captured safely? |
| bxqsq— | Does one or more Black piece control the queening square? |
| dq— | Is there a simple delay to White's queening the pawn? |
| chkmt— | Is the White king in checkmate? |
| stlmt— | Is the White king in stalemate? |
| ds— | Is there a good delayed skewer threat? |

**Example list:**

| **rimmx** | **bxqsq** | **dq** | **chkmt** | **stlmt** | **ds** | **pa7** |
|---|---|---|---|---|---|---|
| no | yes | no | no | no | no | Not |
| no | no | yes | no | no | no | Not |
| no | no | no | no | no | no | Won |
| no | no | no | no | no | yes | Not |
| yes | yes | no | no | no | no | Won |
| no | no | no | no | yes | no | Not |

**Classification rule:**
The RuleFactory representation of the decision tree is on the left. A pseudo-code fragment describing how the rule is evaluated is on the right.

```
rimmx
    no: bxqsq
        no: stlmt
            no: ds
                no: dq
                    no: Won
                    yes: Not
                yes: Not
            yes: Not
        yes: Not
    yes: Won
```

```
if (rimmx = no) then
    if (bxqsq = no) then
        if (stlmt = no) then
            if (ds = no) then
                if (dq = no) then
                    pa7 = Won
                    else pa7 = Not
            else pa7 = Not
        else pa7 = Not
    else pa7 = Not
else pa7 = Won
```

Figure 1. A sample RuleFactory subproblem.

Before RuleFactory can induce a rule for an attribute $A$, the user must supply a list of examples and a list of the attributes which can be used in the rule for $A$. These three things—the classification rule, the example list, and the attribute list—make up a RuleFactory *subproblem* for the attribute. The attribute $A$ is called the *class attribute* for the subproblem. Figure 1 is an example of the contents of a RuleFactory subproblem. (The subproblem was extracted from a chess endgame knowledge base developed by Shapiro [4].)

RuleFactory assumes that all of the attributes it handles at any given time are intended to classify the same sort of thing; i.e. that they all have the same input domain. Such a group of attributes is called a *problem domain.* For example, the attributes which computed some feature of a bridge hand would constitute a problem domain; so would a group of attributes which evaluated a position from a particular chess ending.

## 3. A RULEFACTORY OVERVIEW

In this section we examine the new features which are presently available in RuleFactory or planned for the near future.

### 3.1. The induction module

The induction module is the part of RuleFactory which actually produces the classification rules. RuleFactory will eventually have several induction modules, each using a different algorithm to produce a decision tree from an example set. At present RuleFactory has only one induction module, which is based on an algorithm developed by Quinlan [1, 2] for his ID3 inductive learning program.

The part of ID3 concerned with building a decision-tree is related to Hunt's Concept Learning System [6]. Its purpose is to take a training set $E$ of examples and construct a rule mapping every example in $E$ to its known class value. This is done according to the following three steps:

1. If the set $E$ is empty, then nothing is known about the appropriate class value. The rule produced asserts that the class is the special value *null.*

2. If all of the examples in $E$ have the same class value $C$, then the rule produced asserts $class = C$.

3. Otherwise, the example set contains at least two examples with different class values. We select an attribute $A$ with values $A_1, A_2, \ldots, A_n$, and use this attribute to split the examples into corresponding subsets $E_1, E_2, \ldots, E_n$. The corresponding rule at this point is:

```
if $A = A_1$ then [use rule for $E_1$],
else if $A = A_2$ then [use rule for $E_2$].
...
else if $A = A_n$ then [use rule for $E_n$].
```

Steps 1–3 are repeated recursively on each example subset until each subset is either empty or contains examples with the same class value.

An attribute can only be used once in any path in the tree. If this restriction means that there is no attribute available at step 3, then the attributes in the subproblem are not sufficient to classify the examples. In this case, we say that the example set contains a *clash,* and we form a rule which asserts that the class value is *unknown.*

We now consider the selection of an attribute in step 3 of the main algorithm. Any choice will eventually result in a correct tree, but a series of bad choices will produce a very large tree which treats each example as a special case. A series of optimal choices will produce a minimum-size tree which generalizes over the examples as much as possible.

ID3 uses the entropy measure from information theory in a heuristic for attribute selection. ID3 computes an *entropy score* for each eligible attribute by dividing the examples into the appropriate subsets, computing the entropy of each subset, and totalling the entropy values thus obtained. ID3 then selects the attribute with the *lowest* entropy score. The effect is that at each step ID3 selects the attribute which minimizes the information remaining in the example subsets. This method does not guarantee optimality, but it is relatively fast and in general produces trees which are nearly optimal.

RuleFactory allows the user to control and influence the induction process in several ways. There are five mechanisms in RuleFactory which provide this control:

- manual induction
- partial reinduction
- linear rule constraint
- attribute weights
- reserve examples.

In manual induction mode, the user is responsible for choosing an attribute during the induction process. The program simply displays a list of the available attributes and their entropy scores. The user can duplicate the effect of the induction algorithm by always choosing the attribute with the lowest score. The user may return to automatic mode at any point; the rest of the rule is then formed by the induction algorithm in the normal way. The user may also switch to automatic mode for the remainder of the current subtree, and resume manual mode when the next branch of the rule is entered.

With partial reinduction it is not necessary to create a completely new tree each time the induction routines are used. The user can select any test node in the rule and reinduce the subtree rooted at that node, leaving the rest of the rule intact. The user can also 'mark' a rule node to make it immune to subsequent inductions. The marked node, together with its parent nodes, will not be changed, but children of these nodes will be changed as usual. A special 'reinduce' command ignores marked nodes and creates an entirely new tree.

A linear rule constraint forces the induction algorithm to select from the attributes which are *partial deciders* for the examples in the current subtree (see Arbab and Michie, in this volume, for definitions). If using an attribute would result in decision class nodes for all but one of the branches of the test node, then that attribute is a partial decider for the current subtree. This constraint results in a decision rule that has no branching. RuleFactory will occasionally encounter a situation where the linear rule constraint is in force and yet there are no partial deciders for an example set. The user can relax the constraint and allow this portion of the tree to branch. If the constraint is retained, then RuleFactory gives up and produces a rule which says that the class value for the example set is *undecided.*

When several attributes have the same entropy value, the attribute normally chosen is the one which appears first in the attribute display order for the subproblem. Attribute weights allow the user to override this default tie-breaking scheme. The attribute with the highest weight is chosen out of the group with the same entropy value. Attribute weights are real, non-negative numbers. A weight of zero means that RuleFactory will never choose the attribute, regardless of its entropy score.

The user may place any of the examples in a subproblem into the reserve store. These examples will not be used in the induction process, and will not affect the rule for the subproblem at all.

### 3.2. Structured induction support

When a subproblem for an attribute $A$ contains another attribute $B$ in its attribute list, then we say that $A$ is a *parent* of $B$. RuleFactory supports structured induction by allowing the user to create subproblems for attributes which, like $B$, are used in higher-level subproblems. The *calling structure* of a problem domain is the set of all of the parent/child relations between attributes.

RuleFactory places only one restriction on the attributes which can be used in a subproblem: no attribute can be both an ancestor and a descendant of another. This restriction is designed to prevent recursion during rule evaluation, thus ensuring termination.

### 3.3. Rule-testing and explanation facilities

RuleFactory allows the user to test the rule in any subproblem. When executing the rule, the test module must obtain the value of the attribute in each test node it encounters. If this attribute is a primitive, then the user is asked to supply a value. Otherwise, the rule from the associated subproblem is executed as a subprocedure, returning a value for the attribute. When the execution of the original, top-level rule is complete, RuleFactory presents the computed class value.

RuleFactory can explain the result of a rule evaluation in two ways. One form of explanation is a trace of the exact sequence of attributes tested and values obtained. This describes how the rule evaluation worked its way backwards from the goal (the value of the top-level attribute) to the facts (the attribute values supplied by the user). This explanation form is particularly useful for rule debugging, since it reveals the point of failure in the offending rule. The other type of explanation begins with the input attribute values and shows the intermediate conclusions leading to the goal. Users may find such a 'forward reasoning' explanation to be more understandable, and therefore more convincing.

An example of both forms of explanation is presented in the Appendix.

### 3.4. External attribute classification

In some cases the user may not want to personally supply values for primitive attributes during rule testing; the user might instead prefer to write a function which computes these values. RuleFactory provides a domain-independent interface between itself and the code which the user supplies.

The user begins by designating some or all of the primitive attributes in the domain as *external.* RuleFactory will then expect to obtain their values from an external source and not from the user. The user must then devise a scheme in which every instance in the problem domain is represented by a unique character string, called the *text specification* of the instance. Next, the user writes a C function for each external attribute which accepts a text specification as its input and produces an attribute value as its result. RuleFactory provides a file containing definitions of appropriately named constants; by using these constants in the code the user can ensure a match between the values expected by RuleFactory and the values returned by the functions.

RuleFactory also produces source code for a driver program which takes a request for an attribute value, calls the appropriate function, and passes back the result. The driver program and the user's code are compiled separately and linked together, forming the *external attribute classifier* which is invoked by RuleFactory and used whenever external attribute values are required.

### 3.5. Expert system generation

RuleFactory can produce an expert system for the problem domain in two ways. One of these—the rule-testing facility—is completely automatic and internal. The rule-testing facility in RuleFactory is an expert system answering questions of the form 'what is the value of attribute *A*?' and explaining its reasoning on demand.

RuleFactory can also produce a completely independent expert system for the problem domain. RuleFactory generates a C-coded function for each non-primitive attribute which embodies the associated decision-tree. These functions are then compiled and linked with user interface routines to form the expert system.

The user has several options regarding the independent expert system. The explanation facility is optional, and can be removed to reduce the program size. The functions from the external attribute classifier may be used to provide primitive attribute values if this is desired. The default user interface routines may be replaced with specialized code. In fact, the C code generated by RuleFactory need not be used in an expert system at all; the attribute functions can be included in any program which needs to compute values for the attributes.

## 3.6. Display windows

RuleFactory is designed to work with a standard 80-column terminal. It divides the terminal screen into several sections, with each section displaying some aspect of the current problem domain. Most of these sections, or *windows,* are controlled by the program and are fixed in position. These windows occupy a small portion of the screen. The largest part of the screen is divided into two independent *window frames,* and the content of these frames is controlled by the user. Figure 2 contains a representation of the windows and frames on the RuleFactory terminal screen.

There are three different window types which can be placed in the top and bottom window frames: *attribute windows,* showing the list of attributes used in a subproblem; *example windows,* showing the list of examples used to form the rule for a subproblem; and *rule display windows,* showing the rule formed from the examples. There will always be at least one of these on the screen, called the *active* or *current* window. The subproblem displayed in this window is known as the *cur*ı *ent subproblem.* The user may allow the current window to fill both frames, or may choose to place a second, *alternate* window in one frame.

Certain commands will cause special-purpose windows to be stacked on top of the windows in the user-controlled frames. These windows are created when the command is entered, remain as long as the command is in effect, and vanish (restoring the window in the frame underneath) when the command is completed. The text-editing command creates one of these windows, as does the rule-testing command and the rule-induction command.

The four fixed, program-controlled windows mentioned earlier are shown in Figure 1. The top line of the screen is the *status display window*;

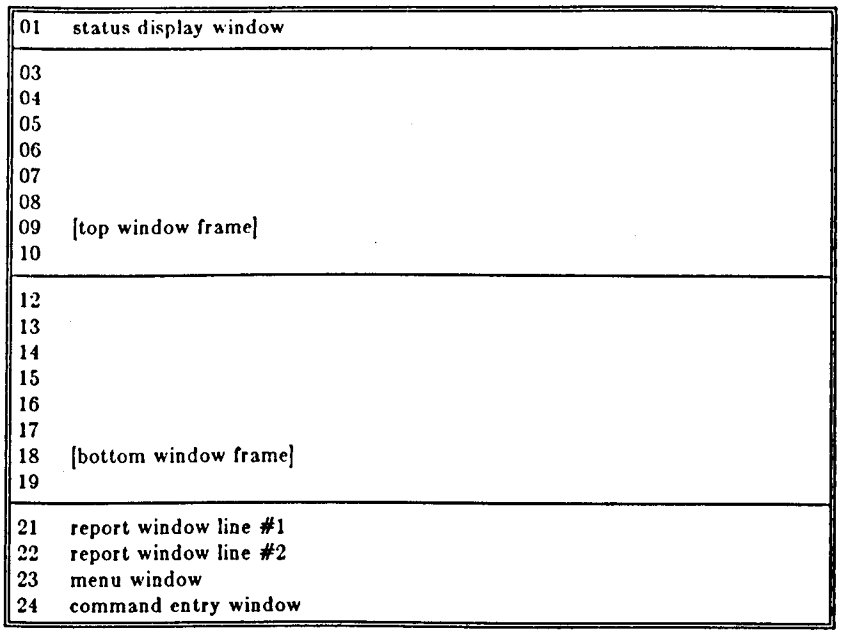


Figure 2. The format of the RuleFactory terminal screen.

it contains the name of the current domain and subproblem, the location of the screen cursor, etc. The bottom line of the screen is the *command entry window,* where the user's input is echoed. Above that is the *menu window,* which contains a list of available commands or otherwise describes the input currently required of the user. Above that is the *report window,* which is used to display error messages and other results of command entry.

## 4. DEFICIENCIES OF THE PRESENT VERSION

The special *unknown* value is returned by a rule when the actual class value cannot be determined from the input attribute values. This result ignores the information which is available in the corresponding example subset, namely, the relative frequency of each class. Out of the examples which match the input attribute values, $n_1$ have class $C_1$, $n_2$ have class $C_2$, etc. If the frequency of the examples in the training set can be made representative of the entire domain, then these class value counts could be used to generate probability estimates for the possible class values. RuleFactory would then return a result stating 'the class value is unknown, but the probability of value $C_1$ is $p_1$, the probability of $C_2$ is $p_2$, etc.'

Sometimes the actual value of a particular attribute is irrelevant to an example. RuleFactory has a special *don't-care* value which can be used in such an example. RuleFactory would profit from the addition of a similar *don't-know* value to be used when an example is true for *some* (but not necessarily all) of the values of an attribute. This would let the user enter one example expressing a general case and several more examples expressing exceptions to the general case without creating a *clash* in the example set.

We could relax the requirement that all attributes must have the same domain by adding a *parameter variable* mechanism to RuleFactory. At present, all attribute functions have one implicit parameter. Adding explicit parameters to the attributes would allow the user to define the domain of each attribute as desired. It would allow the user to define *compound attributes* in a subproblem; for example, in a subproblem with domain $x$, one of the lower-level attributes could be $B(C(x))$. Parameter variables might enable RuleFactory to cope with recursion during rule evaluation, which is currently prohibited.

Finally, RuleFactory needs some form of *probabilistic induction* to deal with cases where there is noise and uncertainty in the input data. This will probably require an extension to the current decision-tree representation of the classification rules.

**Acknowledgment**

This work was in part supported by a research contract awarded to the University of Illinois by Intelligent Terminals Ltd and was supervised by Professors Roy Campbell and Donald Michie.

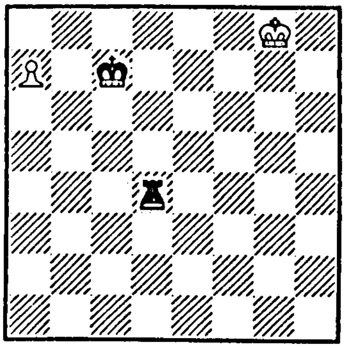
## REFERENCES

1. Quinlan, J. R. (1979) Discovering rules from large collections of examples: a case study. In *Expert systems in the micro electronic age* (ed. D. Michie). Edinburgh University Press, Edinburgh.
2. Quinlan, J. R. (1982) Learning efficient classification procedures and their application to chess end-games, In *Machine learning: an artificial intelligence approach* (eds R. S. Michalski, J. G. Carbonell and T. M. Mitchell), Tioga, Palo Alto, Calif.
3. Shapiro, A. and Niblett, T. (1982) Automatic induction of classification rules for a chess endgame. In *Advances in computer chess 3* (ed. M. R. B. Clark). Pergamon Press, Oxford.
4. Shapiro, A. (1983) The role of structured induction in expert systems. Ph.D. dissertation, University of Edinburgh.
5. Shapiro, A. and Michie, D. (1983) A self-commenting facility for inductively synthesised endgame expertise. In *Advances in computer chess* 4 (ed. D. Beal). Pergamon Press, Oxford.
6. Hunt, E. B., Marin, J., and Stone, P. T. (1986) *Experiments in induction.* Academic Press, New York.

## APPENDIX: EXAMPLES OF RULE EXPLANATIONS

The following examples show how RuleFactory can explain the result of a rule evaluation following the regime of ref. [5]. The problem domain is a

Black



White

particular rook-and-pawn chess endgame; the rule being tested is supposed to determine if a particular position is won for White when White has the next move. This rule was applied to the above position, deciding that the position was not won for White.

## A 'BACKWARD REASONING' EXPLANATION

The first type of explanation is a trace of the exact sequence of attributes tested and values obtained. This form is particularly useful for debugging.

```
Is this position won for White?
  Can the Black rook be captured safely?
  NO
  Does any Black piece control the queening square?
  NO
  Is the White king in stalemate?
  NO
  Is there a good delayed skewer threat?
    Is a special opposition pattern present?
    NO
    Is the White king one square away from the relevant edge?
    NO
  NO
  Is there a simple delay to White's queening the pawn?
    Is there a good delay because there is a mate threat?
      Does the Black rook attack a mating square safely?
      NO
    NO
    Is there a good delay because the White king is on square a8?
```

```
      Is the White king on square a8?
      NO
    NO
    Is there a good delay because the White king is in check?
      Is the White king in check?
      NO
      Is there a good delay because of a double attack threat?
        Is the White king on an edge and not on a8?
        YES
        Is the Black king in the way?
        NO
        Is the Black king attacked in some way by the promoted White
        pawn?
        NO
        Which king controls the intersect point?
        the Black king
      YES
    YES
  NO
```

## A 'FORWARD REASONING' EXPLANATION

The next type of explanation begins with the input attribute values and shows the intermediate conclusions which lead to the final result.

```
Because the Black king controls the intersect point,
  and the Black king is not attacked by the promoted pawn,
  and the Black king is not in the way,
  and the White king is on an edge and not on a8,
there is a good delay because of a double attack threat.

Because the White king is not in check,
there is not a good delay because the White king is in check.

Because the White king is not on square a8,
there is not a good delay because the White king is on square a8.

Because the Black rook does not attack a mating square safely,
there is not a good delay because of a mate threat.

Because there is a good delay because of a double attack threat,
  and there is not a good delay because the White king is in check,
  and there is not a good delay because the White king is on square a8,
  and there is not a good delay because of a mate threat,
there is a simple delay to White's queening the pawn.

Because the White king is not one square away from the relevant edge,
  and there is not a special opposition pattern present,
there is not a good delayed skewer threat.
```

Because there is a simple delay to White's queening the pawn,
  and there is not a good delayed skewer threat,
  and the White king is not in stalemate,
  and no Black piece controls the queening square,
  and the Black rook cannot be captured safely,
the position is not won for White.

## A CONDENSED, 'FORWARD REASONING' EXPLANATION

Finally, the previous explanation can be condensed by discarding the intermediate conclusions and input attribute values which are not directly relevant to the final conclusion.

Because the Black king controls the intersect point,
  and the Black king is not attacked by the promoted pawn,
  and the Black king is not in the way,
  and the White king is on an edge and not on a8,
there is a good delay because of a double attack threat.

Because there is a good delay because of a double attack threat,
there is a simple delay to White's queening the pawn.

Because there is a simple delay to White's queening the pawn,
  and the Black rook cannot be captured safely,
the position is not won for White.

15

# Intelligence Architecture and Inference: VLSI Generalized Associative Memory Devices

D. R. McGregor and J. R. Malone
Department of Computer Science,
University of Strathclyde, UK

**Abstract**
In biological systems intelligence appears as an ability to take advantage of changed conditions in the environment which are of short or very short duration. This speed of response is more important from a survival viewpoint than the ability to form inferences of a novel type, and characterizes lower levels of biological systems (e.g. insects).

In the study of Artificial Intelligence so far, much more attention has been given to the slower learning and inference processes than to the rapid associative recognition processes.

The architecture of biological systems is entirely different from that of current computer or microelectronic devices, and manifestly exhibits vastly superior performance, yet currently there seems to be a taboo against looking to biological systems for inspiration for new VLSI designs.

The paper will describe the design of an ultra-concurrent VLSI generalized associative memory, and progress towards its realization in VLSI.

## 1. INTRODUCTION

Seemingly complex biological behaviour may often be much simpler than it might appear. The first example, originally suggested by Herbert Simon is of a hypothetical ant traversing a pebble beach. The ant's goal is simply to travel straight to its nest—inherently a simple strategy. Because of the pebbles, however, the ant is forced to follow a complex route, and apparently exhibits highly complex behaviour. Simon pointed out that the source of this complexity is in the beach, not in the ant. In fact, the ant is programmed by the beach.

A second example concerns a species of wasp (Sphex). When laying eggs the wasp follows the following routine. First it digs a burrow. Next it captures and paralyses a cricket by stinging it, and drags it to the burrow so that it can act as a store of fresh food for its larvae when they hatch. Before dragging the cricket into the burrow, however, the wasp crawls inside to inspect it, then on emerging, drags the cricket's body inside.

What purposeful intelligence behaviour! If, however, an experimenter moves the cricket's body by only a few centimetres while the wasp is reinspecting the burrow, on emerging she again drags over the body—but then repeats the inspection. However many times the burrow has been inspected, a reinspection is carried out.

It is curious how we as humans regard this repeated behaviour. Our respect for the creature's intelligence turns to contempt—clearly this is not general intelligence, only the reapplication of a simple biological program associated with the immediate context.

Instinctive behaviour of a species is not realized as instinctive behaviour by a member of the species exhibiting it. The routine toilet cleanliness of cats we recognize as 'instinctive'—part of the built-in biological program—but an individual cat will certainly be unaware of this. We suspect that the human species also exhibits types of instinctive behaviour: but individuals if queried invent 'rational' explanations for their actions.

## 2. FAST ASSOCIATIVE RECALL: SLOW INFERENCE PROCESSES

We must not denigrate those parts of the biological intelligence which operate at levels of the intellect beyond the reach of conscious introspection. They may contain the greatest challenge to Artificial Intelligence.

Consider the following two hypothetical school reports:

> Has a good understanding and working knowledge of differential calculus.
> Does well to find his own way home.

Only about a hundred rules are needed to enable a computer system to handle queries concerning differential calculus, but the representation of the knowledge and formation of the plans necessary to do the second are far in advance of the present state of the art. Many of the toughest challenges lie in getting a machine to duplicate the abilities of, say, a squirrel in its abilities to distinguish a branch in a scene with shadows, highlights and leaves obscuring the view, the ability to judge the leap to a branch, and grasp it on landing.

We are clearly influenced in our perception of intelligence by the speed with which a system can respond to stimuli. Time-lapse photography of plants shows that they do respond considerably to light but their real-time response is generally so slow that one would never consider a plant intelligent. Similarly, a computer vision system which takes tens of seconds to analyse a scene seems remarkably unintelligent to the human bystander. An intelligent system requires to function at different time-scales:

—fast associative recall of previously encountered information more complete, but slower search for hypothesis, checks for consistency and so on. We must not expect automatic systems to do these in seconds.

Even to check for logical consistency within a system requires in general an indeterminate amount of computation. Not only is this necessarily slow, it has been shown that it is fundamentally impossible to tell if this process will terminate. (The invention of practical, but limited, checking methods is an important research goal.)

## 3. REPRESENTING KNOWLEDGE: THE FACT SYSTEM [1, 2]

The Fact system developed by our group at Strathclyde is based on a simple representation of knowledge, the elements of which are represented as individual 'molecules' of information. Each 'molecule' is a four-tuple, three fields of which represent a link in a semantic network [3–5] and the fourth is a 'name' or 'label' which can represent the unit of information itself in other 'molecules'. The system is thus capable of representing higher-order logics.

For example, suppose we want to represent the fact that a pen is on the table in front of me. In the Fact system this information is represented by a record with four fields. The first contains a symbol which represents the individual pen in question. The second contains the symbol representing the particular table, while the third field contains a symbol standing for the relationship present. The fourth field represents the unit of information as a whole, and using it we can insert other 'facts' which qualify the first one. For example we can represent the facts that:

'the pen was on the table at 7.30 p.m.'

and so on.

The system may be accessed from both procedural and non-procedural languages, for inserting, deleting, retrieving, and updating information. In addition to the flexibility and representational power which this gives, it allows the development of programs capable of dealing generally with collections of information, irrespective of the subject domain, and data structure.

One such example has been developed which forms hypotheses of a number of general types, which it presents to the user for confirmation. This seems a useful development since human beings are not well adapted to scanning tables of information to try to locate unusual or interesting items of information.

Several types of hypothesis can be formed. This work has general similarities to the work of Lenat at Stanford [6], and Winston at MIT [7], and the work on inductive programming such as that represented in this volume (See also [8, 9]).

## 4. A GENERALIZED ASSOCIATIVE MEMORY DEVICE IMPLEMENTED IN VLSI TECHNOLOGY

Given the superior speed of switching of electronic devices compared with the relatively slow chemical processes in biological systems, why are biological systems so much faster at processing sensory data? The answer lies primarily in their different architectures. Up till now computers have had a serial mode of operation (the so-called Von Neumann architecture) in which only a single operation is obeyed during each machine cycle, whereas biological systems have architectures in which a very large number of processes occur simultaneously. Our own work was motivated by concern at the incompleteness of retrieval of information from conventional systems, and the lack of performance of those which attempted to use inference to overcome this.

The Generic Associative Memory (GAM) device we have developed [10] can be applied to any information handling system. In the context of our own 'Fact' system, however, it can:

1. Evaluate the complete closure of 'semi-explicit' (finite) sets, by stages, or in a single operation, taking approximately 1 ms. This can be used to expand 'is.a' and 'is.subset' hierarchies, and to evaluate closures of other transitive relations. This operation can be applied to find a set of related terms in a stored thesaurus. For example, the expansion of the Class 'Person' would recursively include the expansion of its subsets 'Employee', 'Pensioner', 'Child', and *their* subsets 'Plumber', 'Programmer', 'Manager', etc. down to individual members of these such as 'John Smith', etc. The device can also evaluate the set of all sets to which an entity belongs.

2. Store any desired network, which can be set up dynamically as a transitive, directed graph. A single stored network can be accessed to evaluate the closure of the directed network in a particular direction from each node yielding in general a corresponding number of sets of elements.

3. Rapidly locate the physical storage blocks in which particular elements of data have been stored. This can be used to locate the 'working sets' of information required for a retrieval operation. It may also be used very effectively in 'clustering' data on insertion. Both these are important factors in reducing the 'backing-store bottleneck' of large systems.

4. Be used for high-speed systems such as real-time vision understanding [11].

We are currently pursuing an initial feasibility study in this area in conjunction with Dr Fryer's ESPRIT project concerning the real-time extraction of a 2.5-dimensional sketch.

The device is different from present conventional approaches because:

—it is highly parallel, and extremely fast. Its operation depends on the simultaneous propagation of signals from node to node of a directly switched network. No conventional computation is required, even when evaluating closures. The device takes only a single associative cycle to evaluate a closure of *N* required nodes where a conventional CAM device would require *N* separate cycles.

—as far as we know, this is the first practical implementation of a 'connection machine' architecture in real hardware, using directly switched connections.

Early studies of networks of cells by McCulloch and Pitt, Minsky and others demonstrated the theoretical possibilities of such systems. Later work by Scott Fahlman [5] also demonstrated the attractiveness of the concepts. Recently the approach has received considerable support in the USA with groups at Rochester [12] and at MIT [13].

—The device is tolerant of fabrication faults. An initial self-check selects only fully operational elements for data storage. This combined with the simplicity of the design itself gives good prospects for large devices using Wafer Level Integration, over the next five years.

To see how our device can be used, consider the indexing of conceptually similar information. For example, someone concerned with the 'physiological effect of ethyl alcohol' would certainly be interested in the 'medical after effects of whisky'. The words used can thus be totally different, and may be at different conceptual levels, either more particular or more general. In tackling this problem one must first look

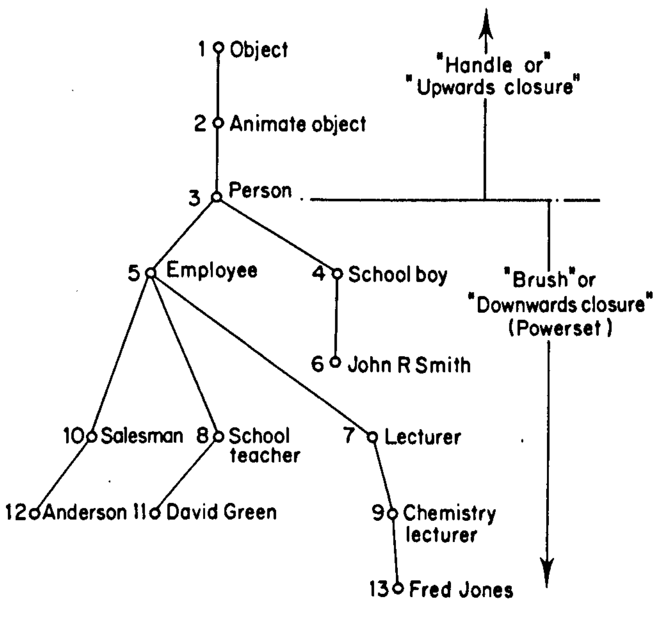


Figure 1.

up the original term in an index, to reveal a number of new terms. In turn each of these new terms must be looked up revealing yet more new terms, and so on. Eventually no further new terms are discovered—we have evaluated the 'complete closure' of the original term, which may well include thousands of terms in total. The VN computer is inherently slow at this operation because it can only look up one term at a time. This was particularly frustrating as we could readily visualize a crude device which could to the job almost instantaneously. This device consists of a permanently wired network (Figure 1) in which each link contains a diode. Simply by applying a voltage to the query node the electric current flows through the network picking out the nodes in the required closure. Unfortunately we would have required a new device for each new set of data.

Our problem was to turn this into a practical, dynamically loadable memory device capable of storing any desired network of transitively linked information. The stages by which this was achieved were: first, to utilize a cross-bar matrix of switches (Figures 2, 3), potentially given *N*

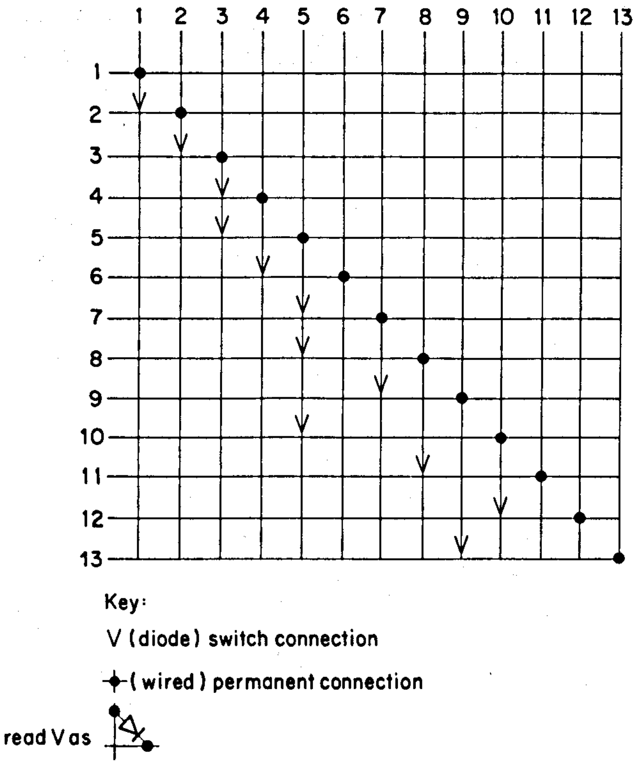


Figure 2.

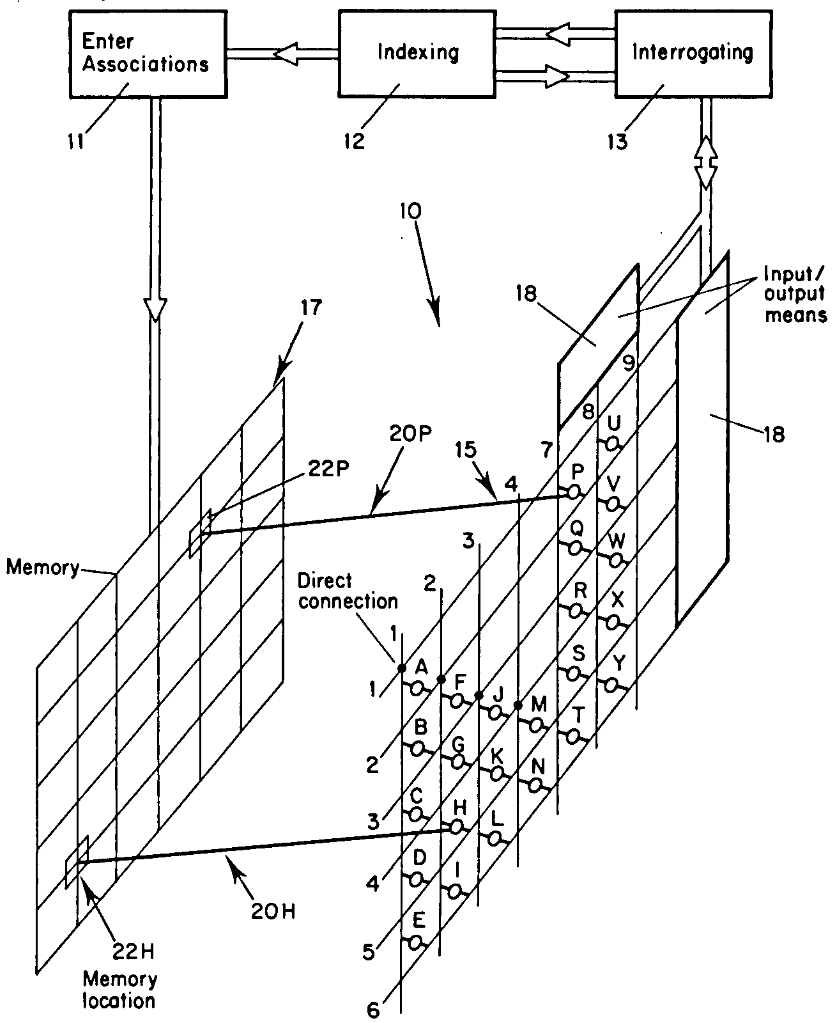


Figure 3.

possible terminals and the possibility of interconnections between any of them, it would appear that $N \times N$ switches are required—a daunting prospect given that we expected $N$ to be in the range 10,000 upwards. Second, provided items are allocated to the memory in such a way that similar items with similar interconnections were stored in neighbouring nodes, it is possible to manage with only a small set of switches on the block-diagonal of the matrix (Figure 5). Figure 4 is a circuit diagram for an $8 \times 8$ matrix.

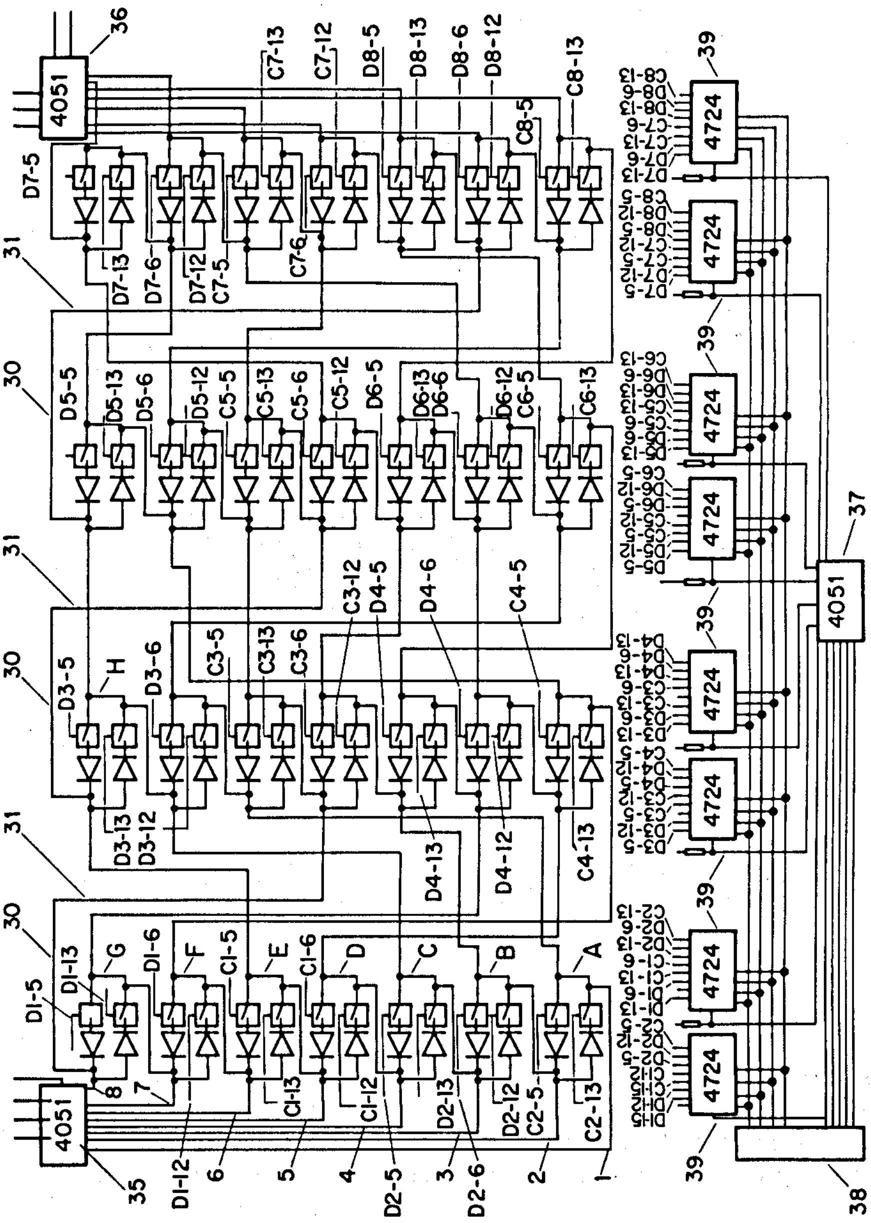

4051
36
35
37
4051
38
39
4724
30
31
A
B
C
D
E
F
G
H
1
2
3
4
5
6
7
8
D1-5
D1-13
D1-6
D1-12
C1-5
C1-6
C1-13
C1-12
D2-5
D2-13
D2-6
D2-12
C2-5
C2-13
D3-5
D3-6
D3-13
D3-12
C3-5
C3-13
C3-6
C3-12
D4-5
D4-6
D4-13
D4-12
C4-5
C4-13
D5-5
D5-13
D5-6
D5-12
C5-5
C5-13
C5-6
C5-12
D6-5
D6-13
D6-6
D6-12
C6-5
C6-13
D7-5
D7-13
D7-6
D7-12
C7-5
C7-6
C7-13
C7-12
D8-5
D8-13
D8-6
D8-12
C8-5
C8-13

Figure 4.

Finally, long-distance connections are handled by a series of wired interconnections between the sub matrices (Figures 5 and 6). The optimum pattern of interconnection of the matrices turns out to be a random wiring: anything else is specialized to some type of data.

These design assumptions were checked by constructing a software simulation of the memory and the complete data base system, and trying it out on a range of problems. Fortunately our design assumptions proved to be correct in practice, and we then proceeded to actual fabrication. The individual matrix components were then designed for fabrication in VLSI.

At the present time, the device has been patented [14], and we are currently awaiting the fabrication of our third version of the chip. Extensive software has been developed, and a company (Deductive Systems Ltd.) has been set up to develop the device for commercial use.

Future developments in VLSI technology are certainly likely to increase the functionality of the device and its storage capacity. By 1990 it may well be possible to store all the associations between say 128,000

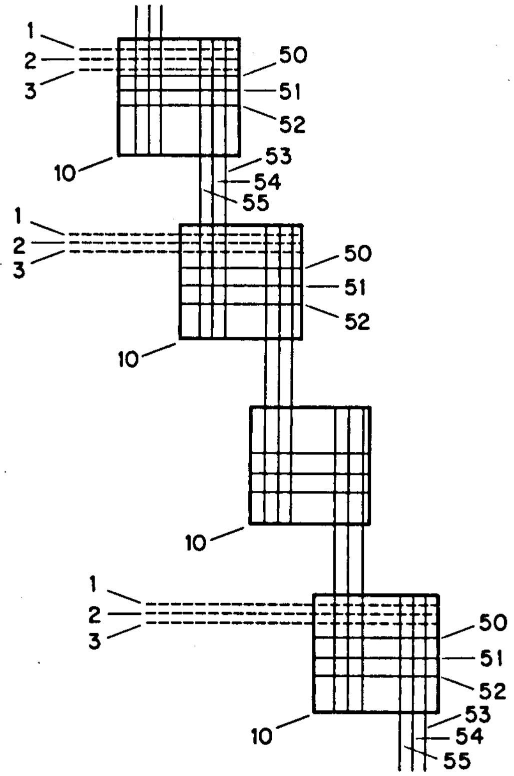


Figure 5.

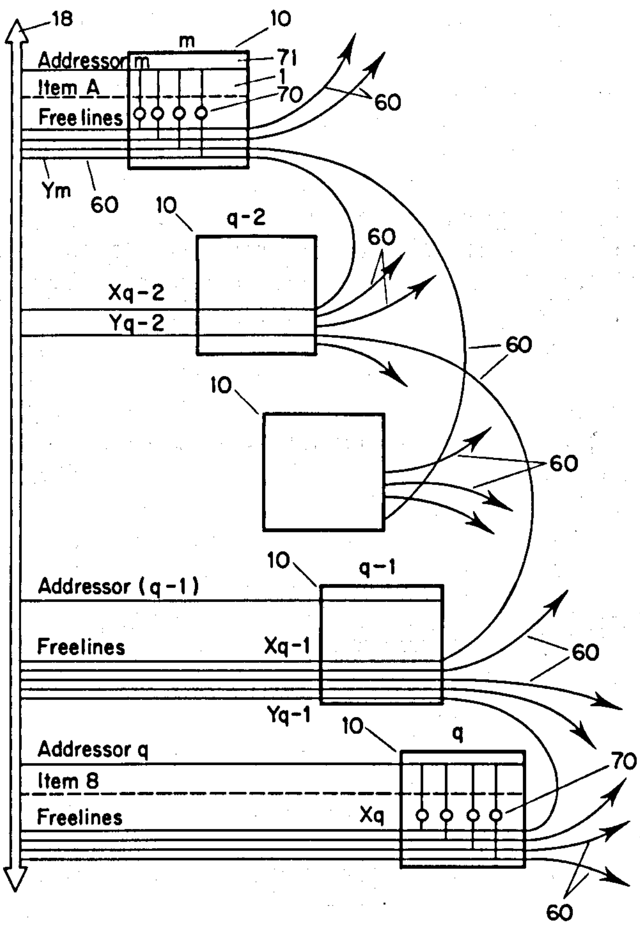


Figure 6.

individual items on a single wafer-integrated component. The time required to associatively recall a set of these will remain short—typically less than 1 ms.

## 5. CONCLUSION

We hope that this paper gives some flavour of the excitement, rapid progress, and enormous potential of the combination of VLSI and AI, though an enormous amount has still be be done before that potential can be realized. It has taken over 80 years to bring automobile design to its present still-evolving form though there have been almost no new technological innovations since 1914. We have barely begun to exploit the microtechnology of the 1970s, while new developments have mushroomed since then and seem likely to continue for the foreseeable future.

## REFERENCES

1. McGregor, D. R. and Malone, J. R. (1982) The Fact database system—a system using generic associative networks. *Research and Development in Information Technology* **1**, 55–72.
2. McGregor, D. R. and Malone, J. R. (1983) The Fact system—a hardware-oriented approach. *DBMS A Technical Comparison—State of the Art Report*, pp. 99–112. Pergamon Infotech, Maidenhead.
3. Barr, A. and Feigenbaum, E. A. (eds) (1981) *The handbook of artificial intelligence*, pp. 180–189. Pitman, London.
4. Deliyani, A. and Kowalski, R. A. (1979) Logic and semantic networks. *Commun. ACM* **22**, 184–193.
5. Falhman, S. E. (1979) *A system for representing and using real-world knowledge.* MIT Press, Cambridge, Mass.
6. Lenat, D. B. (1982) The nature of heuristics. *Artificial Intelligence* **19**, 189–249.
7. Winston, P. H. (1984) *Artificial Intelligence*, 2nd edn. Addison-Wesley, New York.
8. Michalski, R. S. (1983) A theory and methodology of inductive learning. *Artificial Intelligence* **20**, 111–161.
9. Quinlan, J. R. (1979) Induction over large databases, *HPP 79-14*. Heuristic Programming Project, Stanford University, Stanford.
10. McGregor, D. R. and Malone, J. R. (1982) Generic Associative Hardware. Its impact on database systems. *Proc. IEE Colloquium on Associative Methods and Database Engines.*
11. Ballard, D. and Brown, C. (1982) *Computer vision.* Prentice-Hall, London.
12. Feldman, S. A. and Ballard, D. (1981) *Computing with connections TR72*, Department of Computer Science, University of Rochester, N.Y. (1981).
13. Hillis, W. D. (1981) The connection machine, *AI Memo 646*, AI Laboratory, MIT, Cambridge, Mass.
14. McGregor, D. R. and Malone, J. R. (1982) Generic associative memory, G.B. Patent No. 8236084.

# AUTOMATING THE ACQUISITION OF KNOWLEDGE FOR COMPLEX DOMAINS

16

# Expert Against Oracle

A. J. Roycroft*
The Turing Institute,
Glasgow, UK

**Abstract**

A computer-generated combinatorial data base that plays optimally an almost undocumented and very difficult five-man chess endgame (i.e. the data base can be considered as an oracle) was matched against a domain specialist who had prepared for the contest with minimal prior access to the data base. His preparation and strategy are described and the results of the contest itself briefly summarized. The paper closes with an illustrated discussion of the selected endgame in Master practice.

## 1. PURPOSE OF PROJECT

The chess endgame specialist, in contrast to the tournament player, assesses his skill against the criteria of perfection, whether in adjudicating a position or in selecting a move. Automatic construction of complete look-up tables (data bases) by computer makes it possible to apply these exacting criteria in practice. By use of such an 'oracle' Kopec and Niblett (1980) were able empirically to verify the present author's claim to have acquired high-level mastery (in the actual test the play was move-perfect) of the play of won-for-White positions of king and rook against king and knight, an endgame of which full knowledge was lacking prior to the creation of a data base. The purpose of the new project is to investigate the problems and nature of skill-acquisition in an endgame selected as being so complex as to lie beyond the power of the unaided human endgame specialist to master thoroughly. For this purpose the author selected the ending king–bishop–bishop–king–knight (BBN). BBN was historically assumed to be a draw from a general position until at my suggestion Mr Ken Thompson computed an exhaustive BBN data base. The existence of the data base, and samples of its output, were published in the international quarterly magazine *EG* (Thompson and Roycroft, 1983).

* Present address: 17 New Way Road, London NW9 6PL, UK.

The Thompson data base contains complete information on how the two-bishops side can win from all but a few elementary or bizarre positions in a total space of some 250,000,000 positions. It thus exhibits 'skill' at a demonstrably unsurpassable level in a domain that is of formidable difficulty and complexity for human experts.

This characteristic makes possible in principle the absolute measurement of human performance in a difficult domain, and offers the opportunity to explore how 'inert but absolute' knowledge can be accessed and adapted for teaching, for learning, and for the development and validation of expert systems that aim to perform as well as the data base—but without it.

We also for the first time have the possibility to explore thoroughly a significant endgame data base at 'super-expert' level. Future combinatorial data bases in chess and in other domains may be more massive still. Early experience of handling them will be of value.

The present paper reports measurement of the performance of a domain specialist before (Part 1) and after (Part 2) he had been allowed unlimited access to the oracle data base, from which, for any legal position the user can retrieve the following:

(i) all optimal single-ply (that is, at a depth of one white or black move) continuations; and

(ii) the length of the remaining optimal path.

The author is a lifetime student of the chess endgame. He is the author of *The Chess Endgame Study* (1972 and 1981) and edits and publishes the international magazine *EG*. He is a strong, but not master, chessplayer. He is acknowledged worldwide as an endgame specialist.

## 2. THE TASK OF THE DOMAIN SPECIALIST (IN PART 1)

In October 1984 the author undertook:

1. To study the five-man pawnless chess endgame of (white) king and two bishops (one on light squares, one on dark squares) against (black) king and knight, using any available aid except the data base itself, with the exception of a set of 12 variations already excavated and provided in 1983 by Mr Thompson from the data base illustrating optimal play from one of the 32 worst-case-for-White (two bishops) starting positions. Time limit imposed on this period: none.

2. To record as fully and faithfully as possible all thought processes (the dated record to include time taken, chess positions and moves, sources of information used, discoveries, errors, corrections, trains of thought, going over previously trodden ground, etc.).

3. To announce when ready to face the data base, i.e. when no greater mastery of the material seemed achievable by private study.

4. To take the white (two bishops) side against the data base without

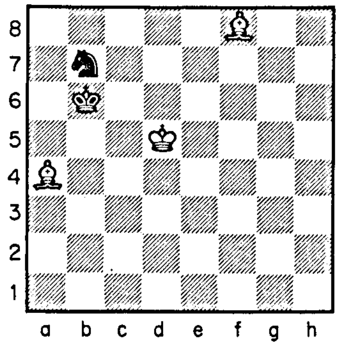
prior preparation of the particular positions to played and to play under strict tournament conditions (moves to be timed, and no moves taken back), with two exceptions: analysis on a separate board to be allowed (the domain specialist is not a practising chess master used to tournament conditions but an endgame scholar accustomed to analysing with board and men, similar to the situation that obtains in correspondence chess); and the so-called '50-move rule' to be disregarded.

The confrontation of domain specialist and oracle data base concludes Part 1. If, as is expected, the human performance is sub-optimal, Part 2 follows, in which the specialist is now allowed access to the data base. A second confrontation or test, with different positions but the same number of them, concludes Part 2.

## 3. BACKGROUND

The earliest known reference in chess literature to the pawnless endgame king and two bishops against king and knight is in the middle of the nineteenth century (Kling and Horwitz, 1851, pp. 62–5). R1, the first of two positions given by the authors, is the more important as it is largely independent of the positioning of the white force. It is given by them without supporting analysis but with the statement that the bishops 'cannot win if the weaker side can obtain a position similar to the above, but they win in most cases'. The second position, a win, is then given with a solution and a number of supporting variations extending to 14 moves. One or other of both positions is repeated in the subsequent literature up to 1983 (e.g. Pachman, 1983, pp. 19–20), with no modification to the verdict.

The author (Roycroft, 1972, p. 207) raised a doubt about the correctness of the claim that R1 (and positions like it) cannot be won. This doubt was confirmed in 1983 by output from the Ken Thompson data base.



R1. Kling and Horwitz (1851). Either side to move.

The data base was generated by a method already known in principle (Ströhlein, 1970). First, all possible positions of checkmate (with the given force), and all positions where the knight is safely captured (without subsequent stalemate), are automatically generated. These comprise the finally won positions that the side with the bishops aims for, and at the same time they are the positions that the side with the knight wishes to postpone as long as possible. From this starting 'position set' the first 'derived set' of positions can be generated, the set of White to Move (WTM) positions that are 'Won in 1'. By an essentially similar, but logically more complex, process the antecedent Black to Move (BTM) position sets are generated and marked where and when appropriate 'Lost in 1'. The basis of an iterative 'maximin' or 'backing-up' procedure has now been established, whereby the solved depth increases in principle by one ply (one white move or one black) per pass. This iteration is initiated and relentlessly pursued until no more positions can be classified. At this stage all won positions will be marked with the solution depth. For a more detailed description of the process see, for example, Roycroft and Niblett (1979) and Thompson (1986). Residual positions still unmarked will be drawn, illegal, or, in a microscopic number of BTM instances, won for the knight's side. In Ken Thompson's solution only WTM positions are stored, the BTM positions being generated when required by program: for convenience we refer simply to the 'data base' whether WTM positions only or both WTM and BTM positions are physically stored.

The results have been widely reprinted in the world's other chess magazines. However, guidance in the domain literature as to how this endgame should be played remains (August, 1985) restricted to paraphrases of the sentence of Kling and Horwitz quoted above, that is that the defending side should always aim for a position like R1, because it is the only safe draw. (At the end of this paper we give an example of the influence of this advice on practical master play.) As a result of the present research it is likely that future advice to the superior side will be to steer towards the Kling and Horwitz position, since the winning method from that position is (or rather will be) well charted. (For a list of the principal authorities on the chess endgame see the entries within parentheses in the section References. However, Averbakh, the major modern authority, does not include the two bishops against knight endgame because endings with two pieces on one side are in principle excluded from its scope.)

## 4. THE FIVE PHASES OF THE PAWNLESS ENDGAME TWO BISHOPS AGAINST KNIGHT

The division of a maximum length solution to this endgame into five phases has been described by the author (Thompson and Roycroft, 1983). The following is an updated version. The quoted passages are taken from the article in *EG*. Where a number of moves is mentioned this refers to consecutive optimal moves by White, the side with the bishops.

### 4.1. Phase number

1. In a maximum depth solution position the white force will initially be under some constraint from the black force: either the white king or a bishop will be immobilized. It may take from six to 12 moves to lift this blockade, depending on its nature.
2. In the next phase Black retreats slowly and in good order and 'seeks refuge in the Kling and Horwitz position. This may be in any corner'. This phase takes us up to move 20, approximately.
3. White's task in phase 3 is to manoeuvre in order to set up any of a small number (probably only four) of 'exit' positions, that is, exits from a Kling and Horwitz position. Black is then forced out into the open. Typically this phase lasts six or seven moves.
4. 'The next stage is complex, fluid, lengthy and difficult. Black strives for maximum freedom, and frequently seems on the verge of achieving it. It takes White some 23 moves, not to be found in any book and characterized at times by excruciating slowness and mystery, before' Black, 'having failed time and again to repeat the Kling and Horwitz position, ends up' with his king 'on the board's edge, near a corner and accompanied by the black knight.'
5. 'The remaining dozen or so moves show the knight being lost, whether he stays close to the black king or runs away.'

The longest solutions have 66 white moves. There are 32 distinct positions that have this depth, though they group into 'families' of positions.

## 5. THE STORY OF PART 1

### 5.1. The 'private study' phase

The private study phase began in October 1984. The material available for study comprised:

(i) published books (in English, German, and Russian) on the chess endgame in general. In contrast to the thriving literature on individual chess openings there is very little published on specific endgames. The books do not cover the endgame two bishops against knight in any useful depth. (See References);

(ii) ten full-length (66 moves) solutions and associated list of one-ply-deep equi-optimal moves provided in August 1983 by Ken Thompson to the author in the latter's capacity as editor of *EG* magazine;

(iii) two further full-length solutions, also from Ken Thompson in 1983, with no alternative moves;

(iv) three 66-move and 67-move full-length solutions from an independent researcher. The 67-move solutions were later shown to be faulty (Comay and Roycroft, 1984), and the correctness of the 66-move maximum optimal depth thereby corroborated;

(v) the 32 distinct positions at the maximum optimal solution length, also provided by Ken Thompson to the author in 1983, but without chess moves;

(vi) the frequency table of the numbers of WTM positions at every solution length from 66 to 1, also provided by Ken Thompson to the author.

On 18th January, 1985 the domain specialist intimated in writing his readiness to confront the data base in the test to end Part 1.

### 5.2. The protocol

Separate publication of the protocol record of the domain specialist's thought processes is intended. It runs to over 200 pages and will be supplemented with appendices.

### 5.3. The Part 1 test and summarized results

The test began on Friday, 29 March 1985. Two test sessions were aborted due to program failure, and there was a two weeks' interruption for holiday. The 10th and final test position was played on Tuesday, 30 April 1985.

Two sessions were abandoned by the domain specialist, after 70 and 69 moves respectively. The remaining eight positions of the test were won by the domain specialist, giving a 'tournament' performance of 80%. The only other measurement of his performance that is available at present to the author is the ratio of the total of the optimal solution depths of the original positions to the number of moves actually taken by the author: 38%.

Both measurements are crude. If the sessions abandoned by the specialist after 70 and 69 moves had been abandoned at the outset without any winning attempt at all, the 38% figure would increase, thereby putting a premium on early abandonment. This is, however, not the case with the refinement (Doran–Michie 'path efficiency') used by Michie (1986) in his review of these same experiments, which rates abandonment at any stage as equivalent to taking infinitely many moves. It is not clear what measurement would be least unsatisfactory. Two other measurements will almost certainly give different figures and should at least be calculated:

1. The ratio of optimal moves to sub-optimal moves in all the moves played by the domain specialist.

2. A measurement that takes into account the domain specialist's division of the endgame into five phases. This would log a minimal penalty against a move that is sub-optimal but which kept the solution within the same phase; it would log a heavier penalty against a move that set the solution back a phase; an even heavier penalty would be imposed on an error that set back the solution two phases, and so on; the heaviest penalty would be for a blunder that gave away the win. If a penalty were measured in numbers of question marks (i.e. '?'), with the lowest penalty rated at a single question mark, then a session could be aborted by prior agreement if the total of accumulated penalties (i.e. the total of question marks deserved) in a session passed a certain threshold. This content-related measurement of performance was proposed by the author but not adopted, one argument against being that the division of solution into phases is at present subjective.

The reason not all measurements are available to the author is that work is proceeding and it is considered that even such ancillary information relating to an earlier test could be of indirect assistance to the domain specialist, who, at the time of writing has not had the test to conclude Part 2. Since a major object of the combined Parts 1 and 2 is to determine how and to what extent a human specialist can be aided in his comprehension of a complex domain, such additional information might, however slightly, distort the performance and measurement. Full statistics will be reported in the planned monograph on the total experiment covering Parts 1 and 2 (Further experiments with the present oracle, and experiments with other, even more massive data bases are envisaged.)

## 6. THE DOMAIN SPECIALIST'S STRATEGIES

Implemented strategies are necessarily domain specific: they have to be described in chess terms. But some general remarks for non-chess players may be helpful.

### 6.1. For non-chessplayers

Here is no place to debate what, if anything, chessplayers have and non-chessplayers lack. But being an amateur problem-solver as well as a chessplayer the author recognizes that all problem-solvers have some common skills and motivations, whatever their specialist knowledge or favoured domain. The awe in which non-chessplayers commonly hold chessplayers of even less than master strength is based partly, if not mainly, on myth. In the interests of better understanding the present section of this paper aims to demolish two specific myths.

*The first myth*: *enormous numbers*

A frequent argument to boost the myth of the arcane genius chessplayer invokes 'enormous numbers'. The number of possible chess positions exceeds the highest astronomical numbers; the number of possible chess games exceeds the number of possible chess positions, also astronomically. These are incontrovertible facts.

But a chessplayer does not have to remember or recognize all these positions and games, any more than any of us need remember or recognize all possible breakfasts in order to eat breakfast, or need remember or recognize all possible books in order to read a book.

Humans tame large numbers by ignoring them. Instead they seek and manipulate patterns, even if the patterns are initially tentative, approximate or unsound. It is a common-sense conjecture of everyday experience that a good pattern will have inferior patterns in its ancestry. If we persevere and are willing to learn, later patterns should be superior to earlier.

In the universal child's game of noughts and crosses (in American it is 'tic-tac-toe', in Russian 'Krestiki-noliki') how many different positions are there? Interestingly, the question is posed, and answers provided, in the literature of artificial intelligence (examples: Nilsson, 1971, pp. 137–8; Shirai and Tsuji, 1982, p. 10; Rich, 1983, p. 7; Alty and Coombs, 1984, p. 80). The usual answer given relies on the implied logic that there are nine initially empty cells to be filled, so we start with nine possibilities for the first play, leaving eight for the reply, seven for the third play, and so on. Factorial 9, or 9! is the (for a game like noughts and crosses, large) number: 362,880—which reduces through laws of legality and symmetry to 'three hundred or so distinct positions with which Nought (by convention the opening player) can be confronted' (Michie, 1961).

A contrasting answer results from arguing that there are three possible states for each cell: a nought, a cross, or emptiness. This gives us a ceiling of three to the ninth power (Rich, 1983), or: 19,683—and this is *before* eliminating symmetries.

Neither of these calculations impresses the *player* of noughts and crosses.

When tackling the identical question he will rather reason like this:

1. There are three rules or conventions that govern the game, and we can look on them as constraints:

—nought starts (constraint no. 1);

—play alternates between the placing of nought and cross (constraint no. 2);

—a completed row (in any direction, including diagonally) ends the game (constraint no. 3).

2. There is then a constraint of a different kind:

—elimination of all symmetries (constraint no. 4).

3. Finally, there is a constraint of a different kind again, one of experience or demonstration, namely:

—a well played game is inevitably drawn (constraint no. 5)

Constraints 1, 2, and 3 are part of the definition of the game. Constraint no. 5 amounts to the constraint of playing the game well. Constraint no. 4 is not essential but it is convenient to all parties.

The player then puts four questions based on the five constraints.

Q1: Can noughts occupy ALL FOUR corner cells?

The answer is 'no', because by constraint no. 5 there is no vacant cell for the fifth nought, needed by constraint no. 2.

Q2: Can noughts occupy just THREE corner cells?

The answer is 'yes', but in only one way.

Q3: Can noughts occupy just TWO corner cells?

The answer here has two parts: if the corner cells are diagonally opposite one another, then the remaining pair of (corner) cells must hold crosses, and any play thereafter into the centre cell will infringe constraint no. 5 again; on the other hand with a pair of cells in adjacent corners it is easy to show (by applying one or more of the five constraints) that there are just two possible distinct configurations.

Q4: Can there be a nought in one corner only or in no corner?

With a nought in just one corner cell, or in none, there is no way to satisfy all five constraints.

The player's answer to the question 'How many positions?' is therefore

1 + 2, i.e. 'three', (See R2.)

We have seen the following answers:

362,880
19,683
3

```
O X O          O X O   O X O
X X O          O X O   O O X
O O X          X O X   X O X
```

Noughts in three corners.    Noughts in two corners.

R2. Noughts-and-crosses/tic-tac-toe/krestik i nulik. The only 'all constraints satisfied' configurations.

Which of them corresponds most closely to the reader's experience of the game?

Chess is more complex (i.e. it holds inordinately more patterns) than noughts and crosses, but humans cannot play chess well without forming, holding and manipulating patterns any more than they can play good noughts and crosses patternlessly. We shall return to this point after demolition of the second myth.

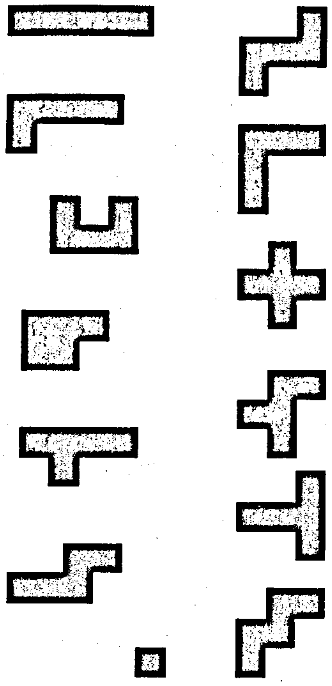
*The second myth: how with all those mobile and differently moving pieces, can chessplayers possibly plan?*

This argument, often in paraphrased form invoking large search trees and high branching factors, requires a different counter-demonstration.

Consider the puzzle of the 'solid pentominoes'. R3 shows the 12 'flat' pentomino shapes, namely all variations on five edge-contiguous identical squares. The solid pentominoes are made out of small cubes instead of squares, but the shapes are otherwise as shown. The puzzle we shall consider is the packing of all 12 shapes into a 3 × 4 × 5 unit dimensioned box. (If preferred, the target shape can be thought of as a 3 × 4 × 5 'brick' to be assembled.) As there are 60 cubelets and 60 spaces to be filled there must be no empty space and no protruding cubelet.

Let us now describe from scratch how this quite tough puzzle might be tackled.

To begin with there is no strategy. We 'play' with the pieces, trying to solve randomly, but as we play we observe ourselves, looking for a pattern. Any discerned pattern is likely to be a pattern of failure, not a pattern of success, but this does not mean that it is not a pattern, nor



R3. the 12 different plane pentominoes that can be formed from juxtaposing five unit squares.

that it will not serve. The pattern which we observe, perhaps, is that after a failed attempt one or more of the shapes in the right-hand column of our figure tend to be unused, and that these are unused more frequently than the shapes in the left-hand column. (In passing, we can ask what the left-over 'awkward' pentominoes have in common with one another? Well, a $3 \times 3$ space (like noughts and crosses—a pattern!) is necessary and sufficient to hold any one of them.) Now from a pattern to a strategy is, for a human, no long journey, albeit not always a conscious or speedy one. In the case of our observed pattern, the derived plan or strategy might be:

> First fit 'several' of the 'awkward' pentominoes together into an *ad hoc* sub-assembly that will 'mutually absorb and minimize' the 'awkwardnesses' of individual pentominoes, and then arrange the remaining 'less awkward' pentominoes around the sub-assembly.

There are two important points about this strategy: it is not precise; and it is readily grasped by a human and implemented by a human, but a strategy it unquestionably is. It lends itself to objective evaluation in an experiment to compare the performance (time taken to solve, success ratios) of two groups of students, one group given the presumed benefit of the 'strategy' and the other group told nothing, but both groups tackling the identical constructional task. A complete set of the 3940 solutions to this puzzle has been computed and published (Bouwkamp, 1967).

That is typical of how the puzzle-solving mind works, whether in chess or pentominoes. Patterns are observed which lead to a strategy. A strategy will not find a solution by itself, but it serves the purpose of enabling the solver to be aware of what he is doing, to have a general aim which he can work with and refine. Such a general aim seems all the more manageable for being imprecise: 'several', 'awkward', 'mutually absorb and minimize', 'less awkward' are fuzzy terms and may be implemented differently (i.e. they may relate to different subsets of the pentominoes or to different subfeatures) by different people, though validly so: there is a distinction to be made between 'relevant fuzziness', which may even be essential in the initial communication of ideas or strategies, and ambiguity, which is to be avoided.

It is the same in chess. If you already have a strategy you can carry it out in your own way, adapting it as you go or discarding it for a better. If you don't, you can't.

If the foregoing account is accepted as valid it follows that for research purposes articulacy in the domain specialist is more important than expertise. A valuable corollary will be that a test of articulacy (to select good human subjects) can be general and standard, though designing such a test is not *a priori* the province of AI researchers.

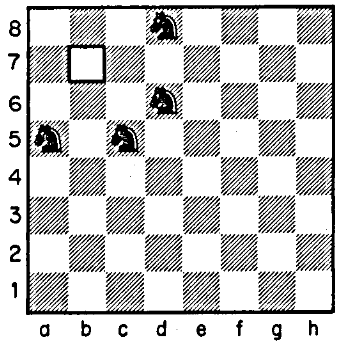
## 6.2. For chessplayers

The problem is strange, even to an experienced chessplayer. Classic concepts, such as the importance of the centre of the board, mobility, sacrificial combinations and positional considerations, turn out to have either no, or limited, application. A long period was spent attempting to apply long-built-in concepts of the classic type, especially those that ought to be applicable to other endgames, but eventually they were largely replaced, or modified, as a result of hard experience.

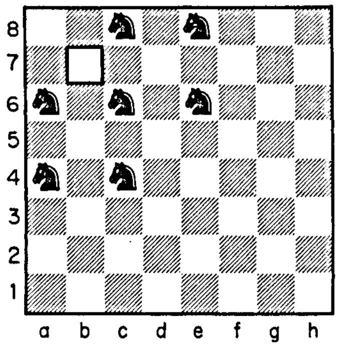
The detailed story will be told in the 200-page protocol. Here only the principal new chess concepts that were found useful will be described. However, one extant concept that proved fundamental and fruitful was the Kling and Horwitz position, though even this tried and tested 133-year-old idea was inadequate in its basic formulation: elaboration was necessary.

## 6.3. New chess concepts

1. *'Knight's distance' from Kling and Horwitz position.* If we consider the square b7, then ONE knight's move's distance means the four specific squares a5, c5, d6, d8. (See R4.) TWO knight's moves' distance means the squares a4, a6, c4, c6, c8, e6, e8, plus the 'less frequent' squares b3, d3, e4, f5, f7 (R5 and R6). The latter five squares are less frequent in the sense that bN is less likely to occupy squares that are in the centre of the board (or in sub-regions controlled by the bishops) because it is the centre of the board that White must and can control in Phase 2 of the contest, the part that consists in driving Black out of the centre. The concept of knight's distance from a Kling and Horwitz position is particularly valuable in the most difficult part of the solution, namely Phase 4. It enables us with confidence (not with certainty) to estimate how far the solution has progressed and to consider particular special strategies and tactics appropriate to that stage, rejecting (i.e. not considering) irrelevant strategies and tactics. Now the four base squares

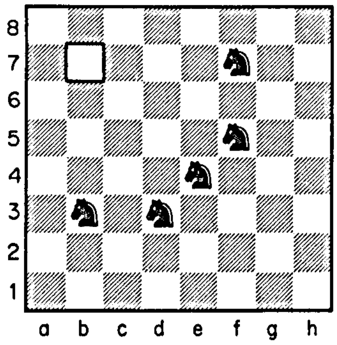


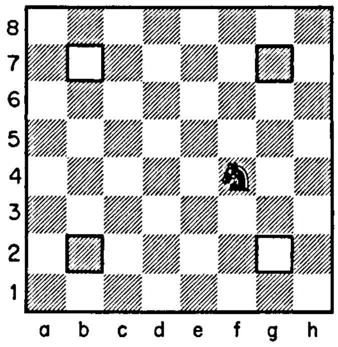
R4. Knight's distance ONE from b7.


R5. Knight's distance TWO from b7 ('frequent' squares).

for Kling and Horwitz position mean different sets of squares in each case. But common squares begin to appear with greater frequency the greater the 'distance number'. Thus the square f4 is ONE from g2, TWO from b2 and g7, and THREE from b7—it is a 'good' square for Black's knight. (See R7.) In practice, since the black king moves slowly, and since the white force will in Phase 4 dominate the centre and some other sub-regions of the board, certain paths will be taboo to the knight: only those Kling and Horwitz positions which are accessible to the black king need be considered in applying the concept of knight's distance from a Kling and Horwitz position.

2. *Black 'king's distance' from a Kling and Horwitz position.* This concept is more static, that is, it is less liable to change from move to move, than the previous concept. The Kling and Horwitz position by definition requires Black's king; Black's king can move only to adjacent squares; it leaps to the eye when Black's king is occupying a Kling and Horwitz square; and distance simply means counting the king moves needed to reach such a square. On f4, for example, the black king is ONE from g3 (for a g2 position) and TWO from g6 (for a g7 position): he is


R6. Knight's distance TWO from b7 ('less frequent' squares).

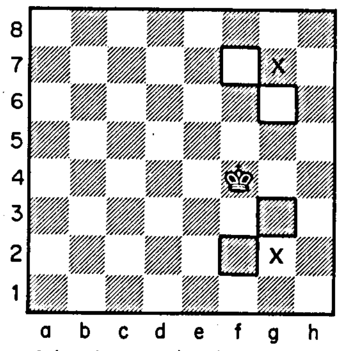


R7. Distance ONE from g2, distance TWO from g7 or b2, distance THREE from b7.

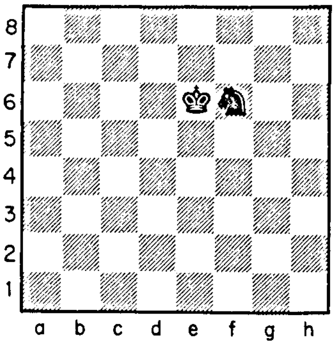
THREE from c2, but this can nearly always be ignored (in Phase 4). (See R8.)

*3. Sum of 'distances'.* This is simply the sum of the black knight's distance from a Kling and Horwitz position and the black king's distance from the same position. (The squares are, of course, different squares: a 'g7' Kling and Horwitz position implies the squares g7 for the black knight and g6 or f7 for the black king.) The summed distances are a rough-and-ready guide to progress in Phase 4. (See R9.)

*4. A 'pseudo-fortress'.* In chess endgame parlance the concept of a 'fortress' is familiar. The implication is that the materially inferior side sets up a position which, due to the geometry of the chessboard is impregnable to the particular attacking potential of the superior side. The edge of the board and especially the corners are suited to fortress positions. A 'pseudo-fortress' arises when Black is evicted from a Kling and Horwitz position (which has been called a fortress, but, we now know, in error) and adopts a posture in which king and knight are alongside each other between two Kling and Horwitz positions (such as



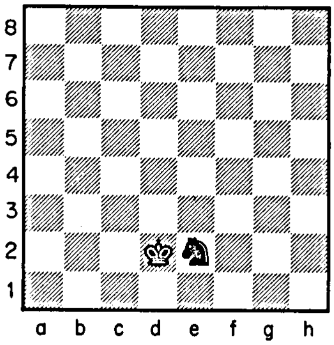
R8. King's distance ONE from g3 (a 'g2' square), King's distance TWO from g6 (a 'g7' square).



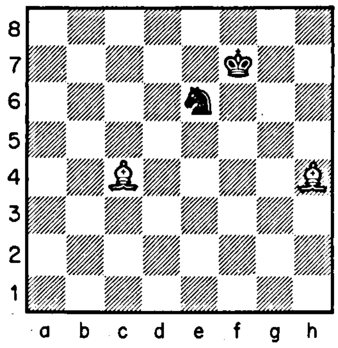
R9. Knight's distance TWO from g7, King's distance ONE from f7. Summed distance: THREE.

between b7 and g7) and at 'summed distances' of THREE or FOUR from each of them. Thus with bKd2 and bNe2 the 'summed distance' is FOUR from b2 and FOUR from g2. (See R10.) We may note, simply for the contrast, that the summed distance is EIGHT to b7 and EIGHT to g7. Such a position is strong for Black because he can adopt the strategy of oscillating to and fro with knight and king without heading for b2 or g2, which White presumably can prevent. Moreover, in his choice of an oscillating move Black, if he cannot check or usefully gain a tempo by attacking a bishop, will tend to choose a move that does not increase the summed distances. There are many manifestations of the pseudo-fortress.

*5. The 'box'.* This is the White 'counter-concept' useful in overcoming the 'pseudo-fortress'. It is a simple idea but its power is best explained by a comparison of chess diagrams and simultaneous consideration of the pseudo-fortress concept. A 'box' is a 2 × 2 array of squares controlled by the pair of bishops. A frequent tactic (in side-variations) to win the knight is a 'pin-crucifix' or a 'checking crucifix', which are simply special cases of the box. (See R11 and R12). Now if Black has a pseudo-fortress,



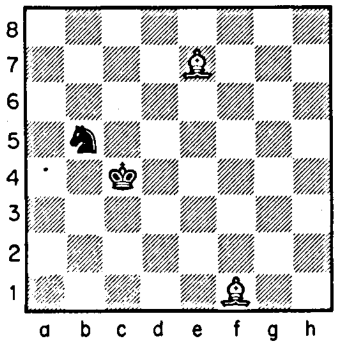
R10. A typical 'pseudo-fortress' with summed distances FOUR from 'b2' or 'g2' positions of the Kling and Horwitz type.



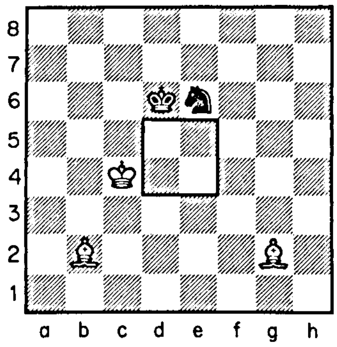
R11. A 'pin-crucifix' using a 'box' e6–f6,e7–f7. The black knight is lost without compensation.

and is happily moving king and/or knight to and fro, making sure to keep the white king at bay by checking when appropriate and then returning to the pseudo-fortress home square, how is he to be evicted? Apart from the king White has only his pair of bishops. They can be used in two obvious ways: on adjacent parallel diagonals from a distance, a classic technique, or by cross-fire to create a box. If we choose a 'box-building' strategy it is not difficult to decide what box is necessary and where the bishops must stand in order to set it up. With the black king on d2 and black knight on e2, the required box will be specified as d2–e2,d3–e3, if we assume that the white king prevents escape to c3. The result of such a box will be to drive the black king towards the cramping edge or a corner without the chance to set up a Kling and Horwitz position. (A box is not a universal panacea. It is a concept to be used with care and cunning.)

*6. 'Advancing' the box.* Place the black king on d6, with the knight alongside on e6—a pseudo-fortress with summed distance THREE to b7 and THREE to g7. Place the white king on c4, and the white bishops on b2

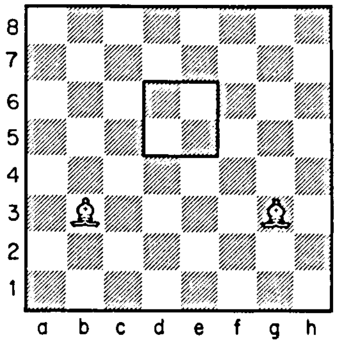


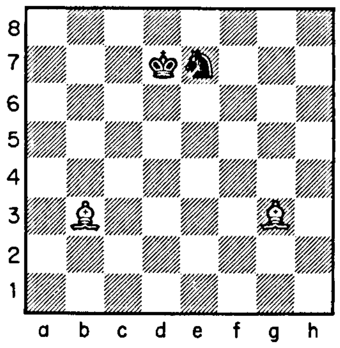
R12. A 'checking crucifix' using a 'box' b4–c4,b5–c5. The black knight is lost without compensation.


R13. Black has a pseudo-fortress. The box d4–e4,d5–e5 is ineffective. The whole box must be shifted one rank up the board.

and g2. (See R13.) The existing box (d4–e4,d5–e5) is a no-man's-land (Black cannot advance further towards the board's centre) but White's task is to drive Black out of his pseudo-fortress. If the chosen method is the box method, then the d4–e4,d5–e5 box must somehow be transformed into a d5–e5,d6–e6 box. To achieve this new box with the bishops that are able to travel only on either white squares or black squares it soon becomes evident that both bishops must switch sides of the board: the dark bishop on b2 must reach g3 (a square on the h2–b8 diagonal) and the light bishop on g2 must reach b3 (a square on the a2–g8 diagonal). (See R14.) This explains otherwise mysterious bishop moves away from the scene of action: they are unavoidable stepping-stones to where the bishops are needed to (threaten to) set up a new box.

7. *'Squinting' bishops.* Place the light bishop on b3 and the dark bishop on g3. Imagine (or place) the black king and knight on d7 and e7 respectively. (See R15.) The white king is centralized, but on no particular square, so it is omitted from the diagram. The bishops are well


R14. A box d5–e5,d6–e6, showing the bishops on opposite sides of the board compared with the d4–e4,d5–e5 box.



R15. The power of 'squinting' bishops (see text).

placed in at least five important respects: firstly, by controlling c7 and f7 they deter the black king from approaching both a b7 Kling and Horwitz and a g7 Kling and Horwitz; secondly, they are immune from tempo-gaining attack by the black king; thirdly, although not immune from attack by the knight they are relatively so, especially as the centralized white king will in Phase 4 control several squares that the knight would need to pass through to execute a bishop-harassing manoeuvre that is frequently a serious threat in Phase 2; fourthly, they create a d5–e5,d6–e6 box; and fifthly they are poised for a quick switch of sides of the board (with the probable, though not necessarily unique, purpose of advancing the box) in at most two moves (each) while still remaining relatively immune from attack (for instance the dark bishop can play to e1 and thence to b4). On the squares b3 and g3 the bishops are not 'glaring straight down' the long diagonals a1–h8 and h1–a8 but are just off-centre in this sense: hence 'squinting'. The term 'cross-eyed' graphically describes the pair of squinting bishops. The set-up is powerful and economical and it occurs frequently, with only occasionally a better square existing: for instance, the dark bishop on g3 might be vulnerable to attack or to a checking fork by the knight playing to f5, with g7 or b7 as possible defensive havens in consequence, and in this case the very remote square h2 might be superior to g3 for the bishop, but only in the short term.

## 7. THE STORY OF PART 2

This is the only section of the current paper to be written after the conclusion of the Part 2 test. Part 2 has no protocol corresponding to Part 1. Instead, dated files were created recording interactive sessions with the data base. Most of the time communing with the data base was spent trying out ideas, alternatives, 'what happens if' hypotheses, derived either from the specialist's own manual record of the Part 1 test, or from attempts at a methodical approach to a problem of a particular phase of play. Considerable time was spent in examination of the output from the interactive sessions. At best such examination would answer a question; at worst it would raise further questions. Close study was made of positions around a solution-depth of 20 moves, with the hope that an increased ability to recognize such long conclusions would ease the understanding of Phase 4, where the really deep play takes place.

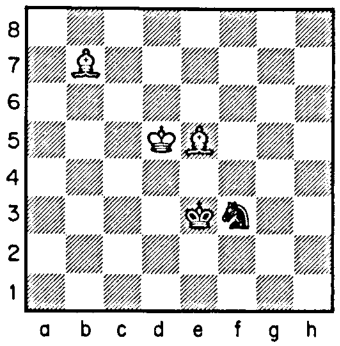
The following are some of the major new patterns to emerge.

1. The black knight is on f3 (or its symmetrical equivalent) with the black king alongside. The knight has two paths to choose from in heading for a g2 Kling and Horwitz. It is extremely unusual for both paths to be blocked. (See R16.)

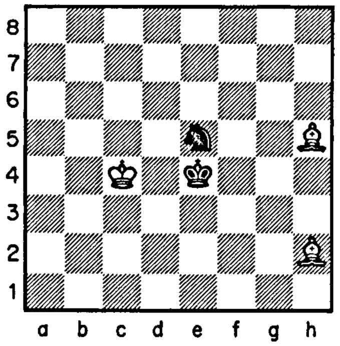
2. The pattern of white king making an outflanking move that avoids checks and covers a presumably important square to free a bishop for more important work. (See R17.)

3. Here there is a box that is prevented by the current placing of the black knight, but it is possible to attack the knight by one of the bishops. The box from Black's standpoint includes a potential outlet for his king, a 'valve', but the same square is where the knight may be lost by a pin-crucifix. This pattern might be dubbed the 'box-valve'. (See R18, R19.)

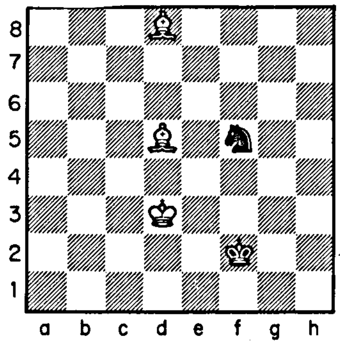
4. In the course of improving the position of the white king it 'follows' his opposite number across the board, keeping to the same or, at worst, adjacent orthogonal, especially during phase 4. In choosing the moment for such a move White must pay special regard to his king's vulnerability to checks from the knight. (See R20.)



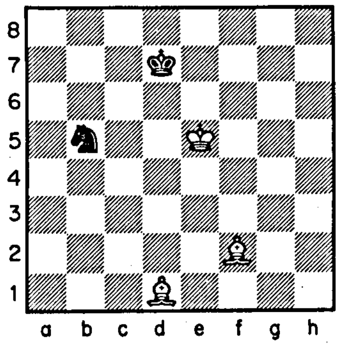
R16. White to move—depth 49. Black has two retreat paths from the square f3 to the Kling and Horwitz square g2: via h4 and via e1. If White prevents this by Bg3, then this bishop has lost its mobility and Black can threaten to set up a b2 Kling and Horwitz position. When Black can obtain a position like this it is a strong indication that the solution depth is near 50.



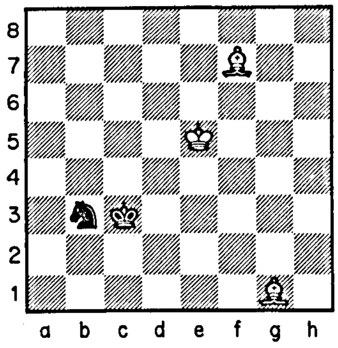
R17. White to move—depth 56. With the sequence Kc5,Nd3+;Kd6, White is safe from immediate checks and prepares to drive the black king towards the edge (any edge) starting with a bishop check on g6.



R18. White to move—depth 13. There is a latent box f3–g3,f2–g2. However, the black knight prevents the box move Bh4+. White plays Be4 and after the knight moves Bh4+ can be played.



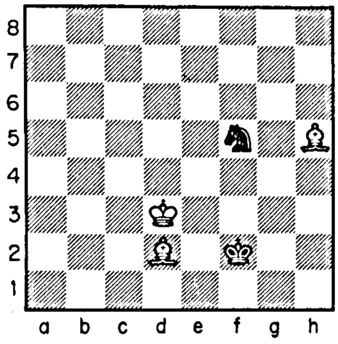
R19. Black to move—depth 19. Black's optimal move is Kc7: which allows White's reply Ke6. The explanation for not choosing Nc3; is that after Bb3,Kc6;Bd4, a box-valve position is created, with the cramping Ba4+ to come.



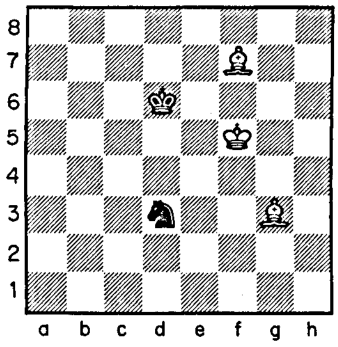
R20. White to move—depth 26. With the move Kd5, the white king is 'following' his opposite number across the board, but occupying central squares in contrast to the black king's more peripheral situation. The move also relieves a bishop of control of an important square, c4 in this case, illustrating the economy of square control by a less mobile piece when the more mobile piece is required for more active work.

5. Black has the occasional strong defence of forcing White to repeat moves as the only alternative to allowing a Kling and Horwitz. Thus a position that has the major feature of phase 4 (Black can be prevented from setting up a Kling and Horwitz) is nevertheless not a win unless progress can be made, and progress, it turns out, is only via phase 3 (a Kling and Horwitz). (See R21.)

6. The black king is forced to block a potential check by his own knight, thus allowing the white king to attack the knight with advantage. (See R22.)



R21. White to move—depth 43. In terms of the Kling and Horwitz prevention heuristic White should control the square h4 by playing Bg5. But Black then renews the threat with Kg3. Then the only way to prevent Nh4; is to play Bd2, as Nh4;Be1+ is good for White. However, in reply to Bd2, Black plays Kf2;, repeating the diagram. White will not make progress by repetition.


R22. Black to move—depth 49. Black is in check and plays his king to c5. This is optimal, despite the fact that it blocks this square for his own knight. It gives White the opportunity to advance his king with a useful gain of tempo by playing Ke4. However, and this illustrates the profundities of this endgame, the move Ke4 is not quite optimal. The sole optimal move is the mysterious Bh2.

### 7.1. Depthcharts

A novel technique for expressing in a concentrated visual form the content of the data base was invented. A diagram was produced showing only four of the five chessmen explicitly. The fifth man was added in the form of a number on each of the squares which the 'missing' man could legally occupy, having regard to which side was presumed to have the move. Such diagrams have been provisionally dubbed 'depthcharts'. A depthchart may identify localities where the depth is (for a white depthchart) significantly low, in which case one may confidently conjecture that if the missing white man is not in that area he ought to head towards it. The technique seems promising, but exploiting it methodically was not possible during the preparation period.

### 7.2. The part 2 test

The second test comprised, like the first, 10 positions.
Number of positions won: seven
Number of positions abandoned: two
One position is unaccounted for: in this, after 105 moves, when the depth of solution was low (nine) a data base move was erroneously executed on the board following the verbal notification over the internal telephone. The consequence was the data base (black) move 'king takes bishop' when the expert's board showed the knight on the square occupied, according to the system, by the black king. This was the only confusion of this kind in all the play of both tests. Performance measured by number of moves played divided by total of optimal lengths: 51.4%.

## 8. THE ENDGAME IN MASTER PRACTICE

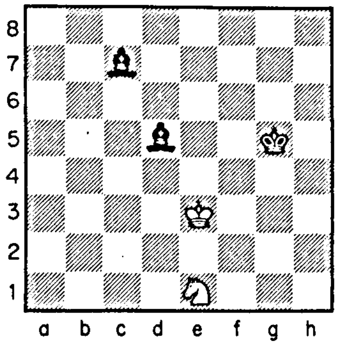
This endgame has occurred several times in the tournament and match practice of the last 25 years but to the best of the author's knowledge the only serious and prolonged attempt to win a deep-solution position took place in the game between International Master Jozsef Pinter (Hungary) and International Grandmaster David Bronstein (USSR) at the international tournament at Budapest in 1978. (See R23; Benko, 1984.)

A comparison of the moves with the optimal moves (obtained by consulting the data base at every step of the game score) is of interest to chessplayers and to non-chessplayers.

Many observations and conjectures are possible arising from the comparison. However, firm conclusions are another matter. We draw none.

All moves played by the two masters from the moment this endgame appeared on the board are given below. The data base assumes that White has the bishops, in accordance with the normal convention of chess endgame theory. Out of respect for the valiant players we keep to the original game colours.

1. The numbering 68–117 corresponds to the serial move numbers of the actual game.
2. Where there is no move in parentheses the played move is optimal.
3. A move in parentheses is optimal, with the implication that the immediately preceding move played in the game is sub-optimal. Note that where there is more than one optimal move, only one is given. In many cases there is only one optimal move, and the occurrence of 12 equi-optimal moves for Black's 95th is unusual.
4. The two-digit number in parentheses after each black move is the optimal depth *after* that move has been played.



R23. Pinter (Hungary) vs. Bronstein (USSR) Budapest, 1978. White to play—position after Black's 67th move.

It follows that optimal play by both sides is identified by consecutive moves without any move in parentheses, when the depth will decrease by one at every move by Black.

The initial WTM position is lost for White in 54 optimal moves.

| White | (optimal) | Black | depth | (optimal) |
|---|---|---|---|---|
| 68.Kd4 | (Nf3+) | Bf7 | (52) | |
| 69.Nd3 | (Ke4) | Kf5 | (50) | |
| 70.Kc3 | (Nc1) | Ke4 | (43) | |
| 71.Nb2 | | Be5+ | (42) | |
| 72.Kc2 | | Bg6 | (42) | (Kd4) |
| 73.Kb3 | | Kd5 | (44) | (Kd4) |
| 74.Na4 | (Nc4) | Kc6 | (49) | (Ke4) |
| 75.Nb2 | (Kc4) | Kb5 | (38) | |
| 76.Nd1 | | Bf7+ | (37) | |
| 77.Kc2 | | Kb4 | (36) | |
| 78.Kd3 | | Bg6+ | (35) | |
| 79.Ke3 | | Kc5 | (45) | (Kc4) |
| 80.Nf2 | (Kf3) | Kd5 | (41) | |
| 81.Nh3 | | Bd4+ | (40) | |
| 82.Kf4 | | Be8 | (48) | (Bg7) |
| 83.Ng5 | | Bb6 | (50) | (Bg7) |
| 84.Nf3 | | Bc7+ | (50) | (Bc6) |
| 85.Ke3 | | Bg3 | (50) | (Bc6) |
| 86.Ng5 | | Bh5 | (49) | |
| 87.Nf3 | | Bc7 | (50) | (Bd6) |
| 88.Kf2 | (Ne1) | Ke4 | (41) | |
| 89.Nh4 | | Kf4 | (40) | |
| 90.Ng2+ | | Ke4 | (41) | (Kg5) |
| 91.Nh4 | | Be8 | (42) | (Kf4) |
| 92.Ng2 | | Bc6 | (43) | (Bh5) |
| 93.Ne1 | | Bb5 | (44) | (Bb7) |
| 94.Ng2 | | Bd6 | (45) | (Bc6) |
| 95.Nh4 | | Kf4 | (46) | (Bc7, and 11 other equally optimal moves) |
| 96.Ng2+ | | Kf5 | (45) | |
| 97.Ne3+ | | Kg5 | (48) | (Ke4) |
| 98.Ng2 | (Nd5) | Bc6 | (43) | |
| 99.Ne1 | | Kg4 | (52) | (Kf5) |
| 100.Ke3 | | Bc5+ | (51) | |
| 101.Kd3 | | Bf2 | (50) | |
| 102.Nc2 | | Kf4 | (49) | |
| 103.Na3 | (Nd4) | Bh4 | (49) | (Be4+) |

| White | (optimal) | Black | depth | (optimal) |
|---|---|---|---|---|
| 104.Kd4 | | Be8 | (48) | |
| 105.Nc4 | | Bf7 | (51) | (Bf2+) |
| 106.Kc5 | (Nd6) | Bf2+ | (33) | |
| 107.Kb4 | | Bd4 | (32) | |
| 108.Kb5 | | Bh5 | (47) | (Be8+) |
| 109.Kc6 | | Bf3+ | (50) | (Ke4) |
| 110.Kc7 | (Kd6) | Kf5 | (46) | |
| 111.Na5 | (Kd6) | Ke6 | (44) | |
| 112.Nb7 | (Nc6) | Ke7 | (45) | (Be5+) |
| 113.Na5 | | Bf2 | (44) | |
| 114.Nc6+ | | Ke6 | (43) | |
| 115.Na5 | (Nd8+) | Bg3+ | (41) | (Ke4) |
| 116.Kb6 | | Bf2+ | (40) | |
| 117.Kc7 | | | | |

At this point the game was declared drawn by the '50-move rule', as no pawn had been moved and no capture had taken place for 50 consecutive moves.

We permit outselves five factual observations and one psychological comment.

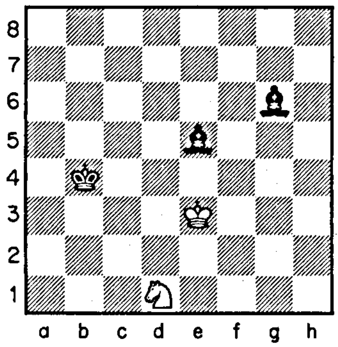
Firstly, it may be seen that Pinter set up the Kling and Horwitz position no fewer than three times, each time in a different corner, namely after his moves 71 (in the b2 area), 90 (in the g2 area) and 112 (in the b7 area).

Secondly, Pinter's knowledge of the Kling and Horwitz position leads him consistently to head for it with unnecessary haste, this accounting for a number of his sub-optimal choices.

Thirdly, the quality of an individual sub-optimal move by either side can be crudely measured (if the opponent's previous and subsequent moves are optimal) by comparing the successive depth numbers. Thus it can be seen that Bronstein's 79...Kc5; increased the optimal depth from 35 to 45, a cruel penalty for not playing the optimal and so similar Kc4; while Pinter's 88.Kf2, reduced the optimal depth from 50 to 41. The great difficulty of this endgame is evident when one tries to give well-grounded reasons for the played moves being inferior to the optimal moves. (See R24.)

Fourthly, an optimal win within the confines of the traditional 50-move rule became possible after White's 70th move, but was lost, never to recur, with Black's 79th.

Fifthly, 36 of White's 50 moves were optimal, while only 26 of Black's



R24. Pinter vs. Bronstein. Position after White's 79th move. Is 79...Kc4; or 79...Kc5; the better move, and why?

49 were optimal. A conjecture is that this is evidence for the endgame being more difficult to play for the side with the bishops.

The psychological comment is that a mistake (as distinct from a crude blunder or oversight) of the kind of Bronstein's 79...Kc5; or Pinter's 88.Kf2, although it leaps to the eye when scanning the depth parentheses to the above game score, is recognized by the player, if at all, only several moves later. The player's general strategy will in all probability have been correct at the highest level, but his ability to calculate in order to reject moves that appear to meet the strategic objectives equally well (which nevertheless fail when countered by an optimal continuation) will be insufficient: one or more vital concepts are missing. Before the creation of the oracle data base no one could have described the feeling for position and depth of calculation needed to play this endgame really well. Now it begins to be possible. The missing concepts are waiting to be formed from data in the data base, and to be verified by reference to the same data base. The idea of *automatic* derivation of concepts or patterns meaningful to human domain specialists is a challenging, a tantalizing, possibility—is its realisation just round the corner or is it remote?

**Acknowledgements**

The 'BBN' data base was generated and generously made available by Mr Ken Thompson of Bell Laboratories, New York, USA. Dr Alen Shapiro of Intelligent Terminals Limited set up the data base in the Turing Institute, added an interactive interface, and was responsible for the admirably smooth conduct of the computing side of the contest. Professor Donald Michie directed the project. I am grateful to The Royal Society and to the Science and Engineering Research Council of Great Britain for the award of an Industrial Fellowship, and to my employer, IBM United Kingdom Limited, for their contribution and associated secondment.

## REFERENCES

Entries entirely within brackets are recognized authorities on chess endgame theory not explicitly referred to in this paper.

Alty, J. I. and Coombs, M. J. (1984) *Expert systems—concepts and examples.* NCC, Manchester.

[Averbakh, Yu. I. (ed.) (1980–1984) *Shakhmatnye okonchaniya.* In five volumes. Fizkul'tura i Sport, Moscow.]

Benko, P. (July 1984) Electronic arts. In *Chess life,* pp. 44–5, 58. United States Chess Federation, New Windsor, NY.

Bouwkamp, C. J. (1967) Catalogue of solutions of the rectangular 3 × 4 × 5 solid pentominoes problem. Technological University Eindhoven and Philips Research Laboratories, Eindhoven.

[Chéron, A. (1960–1970) *Lehr- und Handbuch der Endspiele.* In four volumes. Siegfried Engelhardt, Berlin.]

Comay, O. and Roycroft, A. J. (April 1984) Two bishops against knight (continued from *EG*74). *EG*, no. 75. 249–52.

[Fine, R. (1941) *Basic chess endings.* David McKay, Philadelphia.]

[Hooper, D. V. (1970) *A pocket guide to chess endgames.* Bell, London.]

Kling, J. and Horwitz, D. (1851) *Chess studies, or endings of games.* Skeet, London.

Kopec, D. and Niblett, T. B. (1980) How hard is the play of the king–rook–king–knight ending? In *Advances in Computer Chess 2* (ed. M. R. B. Clarke), pp. 57–81. Edinburgh University Press, Edinburgh.

Michie, D. (1961) Trial and error. *Science Survey,* **2,** 129–45. Reprinted in *On Machine Intelligence* (2nd edn). Ellis Horwood, Chichester (1986).

Michie, D. (1986) The superarticulacy phenomenon in the context of software manufacture. *Proc. R. Soc. Lond.* A **405,** 185–212.

Nilsson, Nils, J. (1971). *Problem-solving methods in artificial intelligence.* McGraw-Hill, New York.

Pachman, L. (1983) *Chess endings for the practical player.* Routledge & Kegan Paul, London.

Rich, E. (1983) *Artificial intelligence.* McGraw-Hill, New York.

Roycroft, A. J. (1972) *Test tube chess.* Faber & Faber, London.

Roycroft, A. J. (1981) *The chess endgame study.* Dover, New York.

Roycroft, A. J. and Niblett, T. (June 1979) How the GBR class 0103 data base was created. *EG* no. 56, 145–6.

Shirai, Y. and Tsuji, J. (1982) *Artificial intelligence.* John Wiley, Chichester.

Ströhlein, T. (1970) *Untersuchungen uber Kombinatorische Spiele.* Technische Hochschule, Munich.

Thompson, K. and Roycroft, A. J. (November 1983) A prophecy fulfilled. *EG* no. 74, 217–20.

Thompson, K. (May 1986) *C* the programs that generate endgame data bases. *EG* no. 83, 2.

# 17 Inductive Acquisition of Chess Strategies

S. H. Muggleton*
Edinburgh University, UK

**Abstract**
A variation of an algorithm for inducing '*k*-contextual' regular language grammars from sample sentences is applied to the construction of expert chess strategies. In a pilot study a small expert system for playing part of the king and two bishops against king and knight endgame (KBBKN) has been automatically constructed using this technique. The generated knowledge-base is directly executable in a MUGOL environment. Although this work is indicative of a new methodology for automatically generating chess-playing strategies from example sequences of play, further work is necessary to show that the technique would be generally applicable to this task.

## 1. INTRODUCTION

### 1.1. Computer chess research

In the study of expert system development, Michie has noted that use of chess expertise as a testbed domain is ideal in many respects. The domain is non-trivial though finitely bounded. It has a wealth of recorded expertise going back many centuries which has certainly not yet been fully exercised. Whereas chess specialists have developed a depth of understanding which is at least comparable with the expertise of more lucrative disciplines, expert-level chess players are generally more readily available for consultation.

Early work in programming computers to play chess was concentrated around efficiently implementing Shannon's chess playing strategy [1]. This employs extensive lookahead in order to compute approximations to the best next move. As this failed to produce results comparable with human expert play, recent research has focused on more knowledge-rich approaches. Bratko and Michie [2] described such a knowledge-based system, AL1, based partly on earlier work by Huberman [3]. AL1's *advice*

* Present address: The Turing Institute, George House, Glasgow, UK.

*module* generated a list of preference-ordered pieces of advice. A separate *search module* used the board-state and advice list to produce a 'forcing tree' which was applied as a strategy for play. As with all solutions in which knowledge must be hand-coded, the knowledge acquisition process becomes a developmental bottleneck.

Quinlan [4] suggested a method of bypassing this bottleneck by using inductive inference. Quinlan's algorithm, ID3, based on Hunt's CLS algorithm [5], was used to build decision trees which classified endgame positions as won, drawn, or lost. A vector of attribute values is used to describe any particular position. This vector together with a class value comprises an example classification. Although the solutions were exhaustively proved correct and ran up to two orders of magnitude faster than commonly used algorithms, they were also completely incomprehensible to chess experts.

In order to circumvent this understandability barrier Shapiro and Niblett [6] introduced the notion of *structured induction,* in which a chess expert is required to decompose hierarchically the endgame classification rules; each subproblem can then be solved inductively. While this approach avoids the problem of incomprehensibility, it unfortunately introduces a new bottleneck of problem structuring.

Paterson [7] has described an attempt to structure automatically the KPK chess endgame domain from example material, using the statistical clustering algorithm CLUSTER. The results, however, have not been very promising, with the machines suggested hierarchy not having any significance to experts. The primary reasons for failure seem to lie in the fact that although the example set is a rich enough source of knowledge to be used for rule construction, additional information is necessary to indicate any higher-level structure.

### 1.2. Sequence induction

In this paper we describe a new approach to the automatic construction of chess strategies from example material. Note that this differs considerably from the approaches of Quinlan [4], and Shapiro and Niblett [6]. In their case, a diagnostic or classificatory expert system was inductively built using static 'snapshot' descriptions. In ours, we build a procedural or strategic expert system from dynamic 'sequence' examples. Each element of the sequence is a snapshot like example of the ID3 variety. The output of the inductive process is a finite state structure in which each state contains a small number of the snapshot examples. These can in turn be used by ID3-like induction schemes to produce rules or decision-trees for each state. *Thus although we do not produce a hierarchical structure, we achieve the aims of structured induction (i.e. a set of small understandable rules) by using example material which contains additional structural information within each example.*

The basis for these techniques lies in the study of grammatical induction, that is the inference of grammatical structures from example sentences of a language [8]. The grammar produced can be viewed as the control structure of a program which generated the example sentences. Some of the earliest work in this area was done by Biermann and Feldman [9] who devised an algorithm to induce finite state automata from strings of a language. Although their algorithm was capable of finding any regular language given a sufficient example set, the algorithm requires an arbitrary complexity parameter and also has rather low *example efficiency* (i.e. a large number of examples are needed to infer anything). Angluin [10] has described an algorithm which infers only a limited subset of the regular languages. This subset she calls the $k$-reversible languages. By limiting the target result set, Angluin's algorithm achieves example efficiency higher than that of Biermann and Feldman's algorithm.

The author [11] has taken Angluin's algorithm and redesigned it to run with $O(n^2)$ time complexity rather than Angluin's original $O(n^3)$ time. Furthermore, we have discovered an even smaller, but useful subset of the $k$-reversible languages, which we call the $k$-contextual languages. The algorithm for inferring members of the $k$-contextual languages is again more example efficient than even Angluin's, to the extent that sensible inference is possible from samples containing only a single example (all other methods in the literature [9, 10, 12, 13] presuppose more than a single example). The $k$-contextual algorithm has $O(n)$ time complexity.

## 2. THE $k$-CONTEXTUAL ALGORITHM

Grammatical induction techniques use exemplary sentences to form generalized grammatical descriptions which are at least compatible with the original example material. Examples can come in two different forms, positive examples and negative examples. Positive examples are members of the target grammar, while negative examples are not. If only positive examples are used then the inductive process must use well-defined constraints on permissible solutions in order to avoid over-generalization. Alternatively, these constraints can be provided by the use of negative examples, in which case, any generated descriptions must hold for all examples that are positive and for none that are negative.

The $k$-contextual algorithm used for the experiments described here requires only positive examples. The necessary constraint on solutions is that the finite state acceptor produced be equivalent to the minimum-size $k$-contextual language containing the positive examples [11]. A regular language $L$ is $k$-contextual if and only if whenever two not necessarily distinct strings $u_1vw_1$ and $u_2vw_2$ are elements of $L$ and $v$ has length $k$, then $u_1vw_2$ and $u_2vw_1$ are also elements of $L$. For normal grammatical

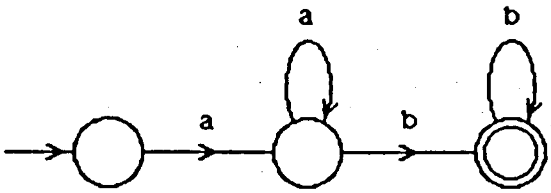


Figure 1. An hypothesized finite state acceptor for the grammatical sample {*aabb*}.

structures, $k$ is a parameter which must be supplied to the algorithm and can be thought of as a complexity measure for the solution. Generally the smaller the value of $k$ the larger the accepted language. However, when dealing with sequences of ID3-like examples, we can use the semantic content provided by the situational vector as an additional constraint mechanism, and thus circumvent the need for supplying the algorithm with the arbitrary measure required by all similar algorithms in the literature [9, 10, 12, 13]. This is achieved by first looking for $k=0$ solutions, and then if a 'clash' (non-determinism) is produced in any of the states of the solution, the value of $k$ is incremented. This process is repeated until either a deterministic solution is produced and the algorithm returns successfully, or it reaches a maximum possible value equal to the maximum length of example string, and returns with failure.

Figure 1 portrays an example of the application of grammatical induction to a set of example sentences (clearly the set contains only one member, i.e. *aabb*). The $k$-contextual algorithm hypothesis represents the language $a^+b^+$.

Situations in which sequence induction can be employed are many and varied [4]. If we understand well what the properties of the algorithm being used are, often we can take advantage of various presentation and solution constraints for different scenarios. Elsewhere [11] several such properties are theorematically described and proved. The most important such property is what Gold [15] calls *identification in the limit.* Let a grammatical induction algorithm $I$ make a hypothesis of a language $L_i$ after each example sentence $u_i$ presented by a complete, arbitrarily ordered enumeration of such examples. $I$ is said to identify the target language $L$ in the limit if and only if there exists some finite natural number $n$ such that $I$ hypothesises the correct language $L_n = L$ following the example $u_n$ and does not subsequently change its guess. The $k$-contextual algorithm used here has been used [11] to identify $k$-contextual languages in the limit.

## 3. THE PROBLEM—KBBKN

Programming strategies for chess endgames is a notoriously difficult task. Zuidema [16] commenting on two Algol 60 programs written for the KRK endgame illustrates the difficulties by noting that 'A small improvement entails a great deal of expense in programming effort and program length. The new rules will have their exceptions too.'

In a project being carried out at the Turing Institute, the extremely complex chess endgame KBBKN is being studied with the aid of the world-class chess endgame specialist John Roycroft. Even this chess authority claims to be out of his depth. In the only definitive study of KBBKN, written in 1851, Horwitz and Kling [17] declared that with White-to-move (WTM), the game is drawn in all but trivial cases. For over a century this claim remained uncontested, until in 1983 Thomson [18] revealed by exhaustive computation that almost all positions are forced wins for White, with a maximum length win of 66 moves being obtainable from 32 different positions [18, 19].

The Turing Institute study involves two phases. In the first, Roycroft has studied the domain intensely with the aim of developing a full descriptive matrix. It is in this first phase that the author has carried out the evaluation of sequence induction as a knowledge acquisition tool. In the second phase it is intended that Roycroft's descriptions be matched against Thomson's exhaustive database for KBBKN.

Roycroft's first task was to select a sub-strategy within the KBBKN domain of an appropriate size and complexity for the application of sequence induction. The choice fell on the first section of the exceptional 66-move forced win for White.

### 3.1. Initial position

Play commences from the position shown in Figure 2.

Taking symmetry and slightly altered starting positions into account, this position is equivalent, in terms of the number of moves to a forced win, to several other similar positions. As this equivalence can be taken into account by the choice of terms in the devised expert system, we will ignore this extra dimension to the problem.

### 3.2. Goal position

The aim of White in this sub-strategy is to liberate wB(light) from the corner in no more than 12 moves. In order to achieve this it is necessary

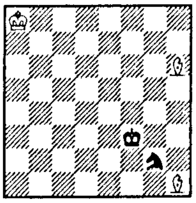


Figure 2. The initial position, WTM.

that

(A) wB(dark) prevents bK from attacking and capturing wBh1. This is illustrated in Figures 3–5.

(B) wK moves to support the attack of wBh1 on bNg2 (Figure 6).

Play achieving (A) is trivially described and encoded. However, attaining (B) is complicated considerably by White's choice of delaying tactics, employed to impede wK approaching h3. It was for this second goal that we use sequence induction to capture Roycroft's description.

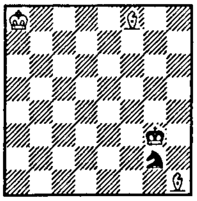


Figure 3. wB(light) prepares to prevent wK from moving to h2, WTM.

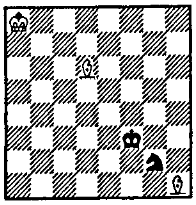


Figure 4. bK retreats after being checked by wB(light), WTM.

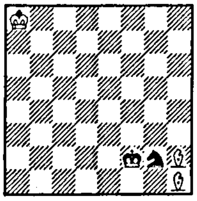


Figure 5. wB(light) takes up fortified position, WTM.

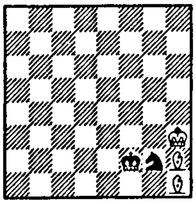


Figure 6. The goal of liberating wB(dark) is achieved, bN is forced to retreat, BTM.

### 3.3. Attributes and actions

Roycroft was asked to give an exposition of play which included a set of sequences of moves together with a running commentary displaying points of interest. From this the author extracted four positional attributes (based on Roycroft's use of adjectival phrases), four actions taken by White (corresponding to verb phrases), and six sequences of play. The attributes were as follows

(B1) Is White free to take bN? {y/n}

(B2) Is wK on the same diagonal as the release position (h3)? {y/n}

(B3) Can wBh1 (dark) move? {y/n}

(B4) Is the direct diagonal position closest to the release position covered? {y/n}

The actions were

(Ba) wK approaches release position (h3) by moving along rank or file.

(Bb) wK moves to non-check position closest to release position on direct diagonal.

(Bc) wB(light) moves out of corner along its diagonal..

(Bd) White takes bN.

Note that each action at this level represents a single move. However, the entire automaton to be derived represents a unit action involving several moves. Thus we might, if necessary, have a hierarchy of such actions and attributes, similar to that described by Shapiro and Niblett [6] for classification (see discussion).

### 3.4. The solution

The sequences used are reproduced in Appendix A. These were presented to a PROLOG-coded version of the *k*-contextual algorithm, the output being translated into a suitable form for further ID3-like induction and run-time testing in the MUGOL environment. Sequences were added by stepwise-refinement, the result being tested after the addition of each sequence. Very early in this process, the *k*-value for the solution rose from 0 to 1, at which level it remained during the rest of development. Also, the number of states in the solution grew rapidly at first to reach a steady value of 5, at which it too stayed fixed. Altogether this process displayed a good incremental nature.

The first six sequences represent White's response to various well-executed tactics played by Black. These were derived directly from Roycroft's description. Having by this stage generated a playing strategy that dealt adequately with more than Roycroft's described positions (the *k*-contextual algorithm successfully generalized solutions to a larger number of positions than those originally described) the automaton was presented and explained to Roycroft. Roycroft noted that the set of positions at which the white king can be delayed by Black was the most complex to describe. *Significantly, the state which described just these positions contained the most ID3-vectors. Thus the structure automatically imposed on the solution had a clear significance to the expert.*

As yet, with only six sequences, the solution was not able to cope with bad play by Black. An additional seven sequences were added to deal with such play. The resulting *k*-contextual automaton is given in Appendix B in a form which can be directly translated into a MUGOL [20] induction file. Appendix C demonstrates the transformation carried out by ID3-like induction to produce a runnable MUGOL expert system. Note that all decision-trees in the solution have the form of HSL [21] decision-trees.

## 4. DISCUSSION

In this paper we have indicated the feasibility of using sequence induction to construct expert-level chess strategies for endgame play. A great deal of further work is necessary to show that:

(i) *optimal* playing strategies can be produced using the technique outlined here;

(ii) solutions generally can be found from *any chosen section* in the KBBKN domain; and

(iii) generated solutions generally are, or can be made to be, *conceptually transparent* to the expert who provided the example material.

For the chosen subgame described here, the methodology used was found by the expert to be natural in terms of the example presentation requirements, as chess players are quite at home with describing play in terms of example move sequences. Furthermore, the bottleneck of structuring was eased, though not completely removed by the use of sequence induction. Whereas other attempts at automatic structuring have led to solutions which are not acceptable to experts, results produced by sequence induction were found to be intuitively correct by the endgame specialist John Roycroft.

In Section 3.3 we noted that as the induced strategy represents a unit action, it might be found necessary to form a hierarchy of such actions in order to create an extensive strategy. Therefore, it might be argued that our automatic structuring aid has gained us no ground, as it may still be necessary to do further manual structuring. We do not claim to have a complete answer to the structuring problem. However, Shapiro [22] in his structured solution of KPa7KR used an average of six examples at each inductive stage in order to produce readable decision-trees. We have used 13 example sequences each containing an average of four ID3-like sequences to produce a semi-structured solution in which each state's rule is derived from an average of only three examples. The example material used here thus consists of approximately $13 \times 4 = 52$ situation/action pairs. Despite the fact that the quantity of example material used to structure this level of problem is an order to magnitude larger than that used by Shapiro, the generated solution contains a small number of easily understandable decision trees.

The *k*-contextual induction algorithm used was found to display good incremental behaviour. This is true in general for this algorithm, which has been proved to identify *k*-contextual solutions in the limit.

On the negative side, we have not developed a form of explanation which deals satisfactorily with sequence execution. It is hoped that by continued research, John Roycroft may be able to suggest a more natural form of explanation in line with that used by chess players to describe sequences of play. Furthermore, since it was necessary to hand-translate ID3-like example material from the PROLOG output form of the *k*-contextual algorithm, it was clear that a better interface to the MUGOL environment is needed.

**Acknowledgements**

This research was carried out while the author was studying for a Ph.D. at Edinburgh University sponsored by the SERC. Thanks are due to the Turing Institute and Edinburgh University AI department for resource and facilities.

We would also like to thank Professor Michie and John Roycroft for their help and guidance, and Thirza A. Castello-Cortes for helping to get it written on time.

## REFERENCES

1. Shannon, C. E. (1950) Programming a computer for playing chess. *Phil. Mag.* **41,** 256–275.
2. Bratko, I. and Michie, D. (1980) A representation of pattern-knowledge in chess endgames. *Advances in computer chess 2* (ed. M. R. B. Clarke) pp. 31–56. Edinburgh University Press, Edinburgh.
3. Huberman, B. J. (1968) A program to play chess end-games. Technical report no. CS 106, Computer Science Department, Stanford University.
4. Quinlan, J. R. (1982) Semi-autonomous acquisition of pattern-based knowledge, *Introductory readings in expert systems,* pp. 192–207. Gordon & Breach, New York.
5. Hunt, E. B., Marin, J., and Stone, P. (1966) *Experiments in induction.* Academic Press, New York.
6. Shapiro, A. and Niblett, T. (1982) Automatic induction of classification rules for a chess endgame. *Advances in computer chess 3* (ed. M. R. B. Clarke) pp. 73–92. Pergamon Press, Oxford.
7. Paterson, A. (1983) An attempt to use CLUSTER to synthesise humanly intelligible subproblems for the KPK chess endgame. University of Illinois series (UIUCDCS-R-83-1156).
8. Muggleton, S. H. (1984) Induction of regular languages from positive examples. *MINews* **5,** 41–59, Turing Institute, Glasgow.
9. Biermann, A. W. and Feldman, J. A. (1972) On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers* **C21,** 592–597.
10. Angluin, D. (1982) Inference of reversible languages. *J. Association for Computing Machines* **29,** 741–756.

11. Muggleton, S. (1986) Inductive acquisition of expert knowledge. Ph.D. thesis, University of Edinburgh.
12. Levine, B. (1982) The use of tree derivatives and a sample support parameter for inferring tree systems. *IEEE Transactions of Pattern Analysis and Machine Intelligence* **PAMI4,** 25–34.
13. Miclet, L. (1980) Regular inference with a tail clustering method. *IEEE Transactions on Systems, Man, Cybernetics* **SMC10,** 737–743.
14. Muggleton, S. (1985) Some experiments with grammatical induction. *MINews* **7,** 60–74.
15. Gold, E. M. (1967) Language identification in the limit. *Inf. Control* **10,** 427–474.
16. Zuidema, C. (1974) Chess, how to program the exceptions? *Afdeling informatica* IW21/74 Mathematisch Centrum, Amsterdam.
17. Horwitz and Kling (1851) *Chess studies*. C. J. Skeet, London.
18. Thompson, K. (1985) Letter to J. Roycroft. *EG* (April 1985).
19. Roycroft J. (1983) A prophesy fulfilled. *EG* (Nov. 1983).
20. Michie, D., Muggleton, S., Riese, C., and Zubrick, S. (1984) RuleMaster: a second-generation knowledge-engineering facility. *Proc. First Conf. on Artificial Intelligence Applications* (1984).
21. Michie, D. (1984) Quality control of induced rule-based programs. *The fifth generation,* CGS Institute notes.
22. Shapiro, A. D. The role of structured induction in expert systems. Ph.D. thesis, University of Edinburgh.

## APPENDIX A—EXAMPLE MOVE SEQUENCES

### Actions

(Ba) wK approaches release position (e.g. h3) by moving along rank or file.

(Bb) wK moves to non-check position on direct diagonal which is closest to release position.

(Bc) wB(light) moves out to corner along its diagonal.

(Bd) white takes bN.

### Attributes

(B1) White free to take bN.

(B2) wK on the same diagonal as release position.

(B3) wBh1 can move.

(B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered).

Sequence 1. Starts from wKa8 and bN does delaying check.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | wKa8 wBh1 wBh2 bKf3 bNg2 | wKb8 |
| n | n | n | n | Ba | wKb8 wBh1 wBh2 bKf2 bNg2 | wKc8 |
| n | y | n | n | Bb | wKc8 wBh1 wBh2 bKf3 bNg2 | wKd7 |
| n | y | n | n | Bb | wKd7 wBh1 wBh2 bKf2 bNg2 | wKe6 |
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Ba | wKf5 wBh1 wBh2 bKf1 bNe3 | wKg5 |
| n | n | n | n | Bb | wKg5 wBh1 wBh2 bKf1 bNg2 | wKg4 |
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf2 bNg2 | wKh3 |
| n | – | y | n | Bc | wKh3 wBh1 wBh2 bKf1 bNf3 | wBa8 |

The '–' in the last line allows the algorithm to generalize to the case in which bN releases wB(light).

Sequence 2. Starts from wKb7 and bN does delaying check.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | wKb7 wBh1 wBh2 bKf2 bNg2 | wKc7 |
| n | n | n | n | Ba | wKc7 wBh1 wBh2 bKf3 bNg2 | wKd7 |
| n | y | n | n | Bb | wKd7 wBh1 wBh2 bKf2 bNg2 | wKe6 |
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Ba | wKf5 wBh1 wBh2 bKf1 bNe3 | wKg5 |
| n | n | n | n | Bb | wKg5 wBh1 wBh2 bKf1 bNg2 | wKg4 |
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf2 bNg2 | wKg3 |
| n | – | y | n | Bc | wKh3 wBh1 wBh2 bKf1 bNf3 | wBa8 |

Sequence 3. Starts from wKb8 and bN does delaying check.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | wKb8 wBh1 wBh2 bKf2 bNg2 | Wkc8 |
| n | y | n | n | Bb | wKc8 wBh1 wBh2 bKf3 bNg2 | wKd7 |
| n | y | n | n | Bb | wKd7 wBh1 wBh2 bKf2 bNg2 | wKe6 |
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Ba | wKf5 wBh1 wBh2 bKf1 bNe3 | wKg5 |
| n | n | n | n | Bb | wKg5 wBh1 wBh2 bKf1 bNg2 | wKg4 |
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf2 bNg2 | wKg3 |
| n | – | y | n | Bc | wKh3 wBh1 wBh2 bKf1 bNf3 | wBa8 |

Sequence 4. Starts with wKa8 and bN does not do delaying check.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | Wka8 wBh1 wBh2 bKf3 bNg2 | wKb8 |
| n | n | n | n | Ba | wKb8 wBh1 wBh2 bKf2 bNg2 | wKc8 |
| n | y | n | n | Bb | wKc8 wBh1 wBh2 bKf3 bNg2 | wKd7 |
| n | y | n | n | Bb | wKd7 wBh1 wBh2 bKf2 bNg2 | wKe6 |
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Bb | wKf5 wBh1 wBh2 bKf2 bNg2 | wKg4 |
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf1 bNg2 | wKh3 |
| n | - | y | n | Bc | wKh3 wBh1 wBh2 bKf2 bNf3 | wBa8 |

Sequence 5. Starts with wKg4.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf1 bNg2 | wKh3 |
| n | – | y | n | Bc | wKh3 wBh1 wBh2 bKf2 bNf3 | wBa8 |

Sequence 6. Starts with wKh3.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | – | y | n | Bc | wKh3 wBh1 wBh2 bKf2 bNf3 | wBa8 |

**Black plays badly**

Sequence 7. Starts with wKa8 after bK has left bN undefended (en prise).

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| y | – | n | n | Bd | wKa8 wBh1 wBh2 bKe2 bNg2 | wB × N! |

Sequence 8. Starts with wKa8 and bK leaves bN as first move.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | wKa8 wBh1 wBh2 bKf3 bNg2 | Wkb8 |
| y | – | n | n | Bd | wKb8 wBh1 wBh2 bKe3 bNg2 | wB × N! |

Sequence 9. Starts with wKg4 and bK leaves bN as first move.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | y | n | n | Bb | wKg4 wBh1 wBh2 bKf1 bNg2 | wKh3 |
| y | – | n | n | Bd | wKh3 wBh1 wBh2 bKe1 bNg2 | wB × N! |

Sequence 10. Starts with wKb8, bN does not do delaying check but allows the release of wB.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | n | n | n | Ba | wKb8 wBh1 wBh2 bKf2 bNg2 | wKc8 |
| n | – | y | n | Bc | wKc8 wBh1 wBh2 bKf2 bNd1 | wBc6 |

Sequence 11. Starts with wKe6, bN does delaying check and then allows the release of wB.

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Ba | wKf5 wBh1 wBh2 bKf1 bNe3 | wKg5 |
| n | – | y | n | Bc | wKg5 wBh1 wBh2 bKf1 bNd1 | wBc6 |

Sequence 12. Starts with wKe6, bN does delaying check and then allows itself to be taken (by moving to g4).

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | y | n | n | Bb | wKe6 wBh1 wBh2 bKf1 bNg2 | wKf5 |
| n | y | n | n | Ba | wKf5 wBh1 wBh2 bKf1 bNe3 | wKg5 |
| y | – | n | n | Bd | wKg5 wBh1 wBh2 bKf1 bNg4 | wKg4! |

Sequence 13. Starts with wKd7, bN checks allowing itself to be taken by wB(dark).

| B1 | B2 | B3 | B4 | Action | Position | Move |
|---|---|---|---|---|---|---|
| n | y | n | n | Bb | wKd7 wBh1 wBh2 bKf2 bNg2 | wKe6 |
| y | y | y | n | Bd | wKe6 wBh1 wBh2 bKf2 bNf4 | B × N |

## APPENDIX B—RESULT OF SEQUENCE INDUCTION

### Actions

(Ba) wK approaches release position (e.g. h3) by moving along rank or file.

(Bb) wK moves to non-check position on direct diagonal which is cloest to release position.

(Bc) wB(light) moves out of corner along its diagonal.

(Bd) White takes bN.

### Attributes

(B1) White free to take bN.

(B2) wK on the same diagonal as release position.

(B3) wBh1 can move.

(B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered)

| B1 | B2 | B3 | B4 | | (Action, Next State) |
|---|---|---|---|---|---|
| State 0 | | | | | |
| n | – | y | n | ⇒ | (Bc, GOAL) |
| n | n | n | n | ⇒ | (Ba, 1) |
| n | y | n | n | ⇒ | (Bb, 2) |
| y | – | n | n | ⇒ | (Bd, GOAL) |
| State 1 | | | | | |
| n | – | y | n | ⇒ | (Bc, GOAL) |
| n | n | n | n | ⇒ | (Ba, 1) |
| n | y | n | n | ⇒ | (Bb, 2) |
| y | – | n | n | ⇒ | (Bd, GOAL) |
| State 2 | | | | | |
| n | – | y | n | ⇒ | (Bc, GOAL) |
| n | y | n | n | ⇒ | (Bb, 2) |
| n | y | n | y | ⇒ | (Ba, 3) |
| y | – | n | n | ⇒ | (Bd, GOAL) |
| y | y | y | n | ⇒ | (Bd, GOAL) |
| State 3 | | | | | |
| n | – | y | n | ⇒ | (Bc, GOAL) |
| n | n | n | n | ⇒ | (Bb, 4) |
| y | – | n | n | ⇒ | (Bd, GOAL) |
| State 4 | | | | | |
| n | y | n | n | ⇒ | (Bb, 2) |

## APPENDIX C—AUTOMATA AFTER ID3-LIKE INDUCTION

### Actions

(Ba) wK approaches release position (e.g. h3) by moving along rank or file.

(Bb) wK moves to non-check position on direct diagonal which is cloest to release position.
(Bc) wB(light) moves out of corner along its diagonal.
(Bd) White takes bN.

**Attributes**

(B1) White free to take bN.
(B2) wK on the same diagonal as release position.
(B3) wBh1 can move
(B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered).

```
State 0
[B1]
          y:⇒(Bd, GOAL)
          n:[B3]
            y:⇒(Bc, GOAL)
            n:[B2]
              y:⇒(Bb, 2)
              n:⇒(Ba, 1)
State 1
[B1]
          y:⇒(Bd, GOAL)
          n:[B3]
            y:⇒(Bc, GOAL)
            n: [B2]
              y:⇒(Bb, 2)
              n:⇒(Ba, 1)
State 2
[B1]
          y:⇒(Bd, GOAL)
          n: [B3]
            y:⇒(Bc, GOAL)
            n:[B4]
              y:⇒(Ba, 3)
              n:⇒(Bb, 2)
State 3
[B1]
          y:⇒(Bd, GOAL)
          n: [B4]
            y:⇒(Bc, GOAL)
            n:⇒(Bb, 4)
State 4
(Bb, 2)
GOAL
```

18

# Validation of a Weather Forecasting Expert System

S. Zubrick*

Radian Corporation,
Austin, Texas, USA

**Abstract**

This paper compares two sources of advice for forecasting of severe thunderstorms: an expert system (WILLARD) and government-issued severe weather outlooks. WILLARD was constructed by a meteorologist using the RuleMaster expert system building facility, which features rule induction from examples of expert decision-making. The validation period spans two months during the peak central United States thunderstorm season for 1984. The forecast comparisons are presented in terms of statistical properties: the Probability of Detection, the False Alarm Rate, and the Critical Skill Index. Even though WILLARD was developed as a demonstration system, its forecasting accuracy on major severe weather days is comparable to government-issued forecasts for the validation period. By examining the results of the comparison, deficiencies in WILLARD were identified that can be rectified in future versions, thereby increasing WILLARD's store of weather knowledge.

## 1. INTRODUCTION

This paper describes the results obtained in comparing two sources of advice for severe thunderstorm forecasting for the central United States (US): one, the standard convective outlook issued by forecasters of the Severe Local Storm Unit (SELS) of the National Weather Service's (NWS's) National Severe Storms Forecast Center (NSSFC); the other, a similar outlook made by a prototype expert system called WILLARD.

A thunderstorm is considered severe if any one of the following phenomena accompanies the thunderstorm (and is reported):

- tornadoes (intense, small-scale cyclones);
- hailstones $\geq 2$ cm ($\frac{3}{4}$ in.) in diameter;
- surface wind gusts in excess of 93 km h$^{-1}$ (50 knots) and/or significant wind damage.

* Current address: National Oceanic and Atmospheric Administration, National Weather Service, Silver Spring, Maryland, USA.

To understand better the nature of severe thunderstorm forecasting. Section 2 discusses some fundamental reasoning processes used by a meteorologist in forecasting severe thunderstorms. An overview of the method used by SELS forecasters at NSSFC will be provided, followed by a description of the WILLARD expert system.

Testing WILLARD using actual weather data is the subject of Section 3, which includes a discussion of the verification methods used. Also included are definitions for and the significance of three statistical parameters important in assessing a severe thunderstorm forecast: the Probability of Detection (*PoD*), the Critical Skill Index (*CSI*), and the False Alarm Ratio (*FAR*). These will be given for WILLARD's and SELS' outlooks. Other statistical parameters useful in verifying forecasts are also presented, together with comparison and discussion of forecasts made by SELS and WILLARD for selected days during the springtime 1984 central US severe thunderstorm season. The statistical parameters introduced in Section 3 for WILLARD's outlooks are shown to be comparable to NSSFC forecast outlooks in many respects. WILLARD's forecast reasoning is compared with that of SELS for a cross-section of severe, marginally severe, and non-severe weather days. An appendix for each test case study day highlights relevant meteorological factors recognized by each forecaster (SELS and WILLARD).

## 2. PROBLEM DEFINITION

'*Successful tornado and severe-thunderstorm forecasting is largely dependent upon the forecaster's ability to carefully analyze, coordinate, and assess the relative values of a multitude of meteorological variables, and mentally integrate and project these variables three-dimensionally in space and time.*' (Miller, 1972, p. 1.)

### 2.1. Weather data and forecasting

An important forecasting task is evaluating the basic accuracy and reliability of meteorological data. The forecaster must have physical access to weather data with enough time to analyse them and make his forecast. Much weather data are contained in graphical format, such as maps, charts, and satellite images. Most time spent by the forecaster in preparing his forecast is used in carefully examining and analysing these graphical data. Pattern recognition is an important technique the forecaster uses in extracting features important in severe thunderstorm forecasting. In fact, recent advances in severe thunderstorm forecasting accuracy have been attributed to use of sophisticated graphic display workstations that allow forecasters to examine quickly a large amount of meteorological data (Suomi *et al.*, 1983; Reynolds, 1983; Kerr, 1984; Mandics and Brown, 1985).

A fundamental factor affecting severe thunderstorm forecast accuracy involves the temporal and spatial resolving power of the current data-gathering network in the central US. This network was originally designed to observe large-scale weather systems: the major cyclones and anticyclones that govern regional weather and the fronts between air masses of different properties. In the central US, stations taking surface observations are spaced roughly 100–250 km apart and measure surface conditions hourly. Stations which measure winds, temperatures, and moisture through the depth of the atmosphere are spaced even further apart; averaging 300–500 km apart and taking observations only twice per day.

As such, this network does a good job providing information concerning these large-scale systems. However, it provides only sporadic information about individual thunderstorms and tornados which occur on the 'storm-scale' (i.e. between 2–200 km; 0.5–6 hours) between observing stations, and very little information about the specific location and time of occurrence of these severe events. This affects severe thunderstorm forecasting because there is insufficient resolution of weather features that play major roles in delineating severe areas.

Since the data network is not equipped to observe storm-scale weather phenomena, the forecaster must squeeze out small details from data sets to make his forecast of severe weather, often basing his forecast on somewhat shaky evidence obtained from the data. Solutions have been offered by respected institutions which would effectively increase the amount of data available from the network 10-fold (UCAR, 1982). One wonders if the forecasters will be able to assimilate this increase effectively. However, it is hoped that expert systems for weather data assimilation and forecast guidance support might go some way towards solving this problem in the future.

### 2.2. Human forecaster skills

Severe thunderstorm forecasting requires the ability to integrate and process an enormous amount of weather information contained in the data to produce a forecast within a short time period. In many instances a parameter overlooked by a forecaster because he is too busy could significantly affect the placement of a severe weather threat area.

Another factor affecting forecast accuracy is the forecaster's ability to recall his knowledge accurately and consistently. Forecasts have often been wrong simply because the forecaster forgot a general heuristic rule governing a particular severe weather situation. This occurs more frequently at the beginning of the peak severe thunderstorm season, when forecasters tend to be a little rusty in applying their rules. In the central US, the peak severe thunderstorm season runs from March through July.

WILLARD was designed as an expert system to aid severe thunderstorm forecasters to improve their ability to forecast severe weather in a more accurate and timely manner, by: (i) providing a consistent expert-forecast knowledge base to the forecaster; and (ii) routinely applying this knowledge base to incoming data to yield initial guidance available in real time.

Before discussing how WILLARD operates, this paper will discuss the basic ingredients for producing severe thunderstorms, and introduce some basic terminology. This is necessary for understanding the discussion of comparisons between WILLARD's forecast advice and advice given in government-issued forecasts.

### 2.3. Severe thunderstorm forecasting methods

It is beyond the scope of this paper to explain all details involved with severe thunderstorm forecasting. There are a few 'cookbook-style' texts that offer fairly explicit methods (see Miller, 1972; Crisp, 1979). The meteorological literature contains a wealth of information (see, e.g., Foster and Bates, 1956; Maddox and Doswell, 1982; Porter *et al.*, 1955). This paper provides a cursory introduction to severe thunderstorm forecasting to familiarize the reader with common parameter names. It will briefly discuss three ingredients necessary for severe thunderstorms: moisture (convective instability), lifting (triggering) mechanisms, and venting mechanisms (which also act as trigger mechanisms).

#### 2.3.1. *Moisture (convective instability)*

It is not surprising that moisture is a key parameter analysed by severe thunderstorm forecasters. When moist air ascends it cools and allows condensation of its water vapour to form clouds. But without a mechanism that allows moist air to be rapidly carried upwards, causing an explosive release of an air parcel's latent heat of vaporization to the surrounding environment, there would be no severe thunderstorms.

Most severe thunderstorms are associated with areas where the convergence, or focusing, of moisture in the lowest few kilometers of the atmosphere is concentrated over a relatively small area (several thousand square kilometers ) (Hudson, 1971). Moisture convergence zones tend to be found along thermal boundaries, like warm and cold fronts. These zones of concentrated low-level moisture are the favoured breeding grounds for severe thunderstorms.

In the central US, this moisture usually originates from the Gulf of Mexico and is normally fairly warm. This warmth also aids in thunderstorm formation. In severe situations, moisture is often found in a distinct tongue of high moisture that is rapidly carried northward into the interior of the central US. An unstable air mass is one that is both warm and moist. If lifted, air parcels within an unstable air mass easily become buoyant and aid the growth of thunderstorms. This latter concept is referred to as *convective instability*. There are various measures of air mass instability. Some common ones are the Lifted Index (Galway, 1956), *K*-Index, and Total–Totals Index.

### 2.3.2. *Lifting mechanisms*

Once an area of low-level moisture convergence is found, one looks for a suitable mechanism that allows the moist air mass to be lifted aloft. These mechanisms are generally characteristic of atmospheric features called upper-level, low-pressure centres or troughs. In fact, a low-pressure trough is akin to a trough in a water wave. Forecasters look for severe weather in front of an approaching low-pressure trough because this region provides an environment of rising air motions.

Movement of upper-level, low-pressure troughs are monitored by examining a parameter that measures the spin of the atmosphere known as vorticity. Regions where upper-level vorticity is a maximum are highly correlated with maximum rising air currents.

Other mechanisms that allow for rapidly rising air currents include strong surface daytime heating (causing moist air parcels to be intensely buoyant), orographic flow (as in flow rising over a mountain), frontal boundaries (which provide mechanical lift from cold (dense) air wedging underneath warm, moist (less dense) air, and small scale surface circulation features known as meso-lows.

### 2.3.3. *Venting mechanisms*

After the moisture field and lifting mechanisms are identified, the forecaster then determines if there is present an upper-level feature that essentially acts as a 'venting mechanism'. This mechanism allows rising air currents to be carried up and out of the lower atmosphere. If strong enough, this sets up a vertical circulation that intensifies the thunderstorm (McNulty, 1978).

Upper-level venting mechanisms are found when the forecaster spots an extremely fast and narrow upper level wind current (or jet streak) flowing at about 10 km in altitude. This high-speed wind current tends to draw air upwards through the storm's centre—akin to a high wind drawing air up a chimney. This results in increased amounts of warm moist air being drawn into the storm by the low-level winds converging into the storm system. The higher this jet streak's speed, the more destructive are the severe storms. Speeds observed in violent severe storm systems range anywhere from 250 to over 400 km $h^{-1}$.

In the central US, it is common in the springtime for this high-speed wind current to be associated with the subtropical jet stream, which is frequently associated with severe thunderstorm outbreaks (Whitney, 1977). Certain regions surrounding the upper-level jet max core are conducive to enhancing severe weather, especially when the exit region of the jet streak interacts with lower-level wind features to create differential temperature and moisture transports (Uccellini and Johnson, 1979). The presence of a strong upper-level jet streak is highly correlated with the formation of strong tornadoes.

Generally, for severe thunderstorms to occur there needs to be a 'phasing in' of maximum low-level moisture convergence, lifting mechanisms, and venting mechanisms over a small region. The major forecast objective is to identify these small regions and define the times of severe weather onset and cessation.

### 2.4. Severe thunderstorm outlooks

#### 2.4.1. *Government-issued severe thunderstorm outlooks*

The SELS of the NSSFC issues medium-range, severe local storm outlooks three times daily during the US severe thunderstorm season. An early outlook is issued at 08.00 Universal Coordinated Time (UCT or Z), a morning outlook is issued at 15.00Z and an afternoon update is issued at 19.30Z. These outlooks are disseminated to various government and private agencies to provide preliminary guidance on expected severe local thunderstorm development in an 18–24 time period covering the entire contiguous US. For example, the outlooks are used by NWS Regional Forecast Offices in preparing state, zone, local, and aviation forecasts (Otsby, 1979).

A severe thunderstorm outlook contains a phrase specifying the expected areal density of severe weather coverage occurring within the valid period of the forecast area. A typical SELS outlook covers an area approximately 337,000 square kilometres [130,000 square statute miles (sq sm)]. The areal density/risk categories as specified in the NWS Operations Manual Chapter C-40 (1979) that are followed by SELS forecasters are:

- Slight risk: 2–5 per cent areal coverage or 4–10 MDR (Manually Digitized Radar, see below) blocks with severe thunderstorms per 100,000 sq sm of outlook;
- Moderate risk: 6–10 per cent areal coverage or 11–21 MDR blocks with severe thunderstorms per 100,000 sq sm of outlook;
- High risk: greater than 10 per cent areal coverage or more than 21 MDR blocks with severe thunderstorms per 100,000 sq sm of outlook.

An MDR (Manually Digitized Radar) block measures approximately 41 km (22 sm) on a side, occupying an area of 1681 sq km (484 sq sm).

Note that only a small fraction of the outlook area is expected to experience severe weather. Other thunderstorm categories are given in a SELS outlook. They include an approaching density/risk and a general non-severe thunderstorm category. However, both of these categories are classified as non-severe and are not verified at SELS so they will not be considered in this paper.

Each SELS outlook contains forecast reasoning in the form of a narrative text. This reasoning points out the major factors that influenced the forecaster's selection of a threat area. It basically relates how present weather conditions will evolve to allow the formation of severe thunderstorms. This reasoning is used by forecasters in the field either to accept or to modify the outlook area. Graphic maps are also disseminated over facsimile circuits for SELS outlook areas.

In producing the outlook, SELS forecasters have considerable prognostic and diagnostic guidance available (Pearson and Weiss, 1979). Prognostic guidance is derived in part from an operational numerical primitive equation model of the atmosphere called the Limited Fine Mesh (LFM) II Model.

From LFM data, contour maps are generated and disseminated to SELS forecasters who examine these maps for clues of impending severe weather. The same LFM data used to generate these maps is used in its gridded form by a host of FORTRAN analysis routines callable by WILLARD. These routines are used to extract information on features necessary in producing the severe weather outlook.

Diagnostic aids used at SELS include computer-plotted surface weather maps, upper-air soundings, and numerous derived objective analyses, such as 500 millibar absolute vorticity, upper tropospheric mean divergence, air mass stability, and low-level moisture convergence. In addition, the latest visible and infrared satellite imagery is available to forecasters for evaluating and updating the numerical guidance. Other members of the NSSFC staff assist SELS forecasters in analysis of local radar data for identifying severe thunderstorms. (Satellite data was not used in WILLARD.)

This paper will examine (for comparison purposes) the forecast outlooks issued by SELS at 08.00Z. This outlook relies heavily upon numerical guidance from the LFM model, especially the 24-h model forecasts made from the 00.00Z LFM model run cycle. The forecaster usually has at least 3–5 h available to examine model results before issuing the 08.00Z outlook.

### 2.4.2. *WILLARD: an expert system to produce severe outlooks*

The expert system WILLARD produces an outlook of severe thunderstorms that is similar to the SELS 08.00Z convective outlook. However, it differs from the SELS outlooks in that its areal extent only covers the central one-third of the US (Figure 1), and is valid for a 12-h time period rather than a 24-h period.

Figure 2 outlines the general information flow path while running WILLARD. The main source of input data used by WILLARD (viz., the LFM model output forecast data) is a major subset of input data used by SELS forecasters in generating their 08.00Z severe outlook.

To produce an outlook with WILLARD, gridded data from the LFM model is obtained and stored on the computer (WILLARD runs on a Sun 100). A host of FORTRAN analysis routines that compute various diagnostic parameters from the LFM data files are available to WILLARD. When WILLARD's knowledge-base is run, it obtains most of the necessary answers to its questions by requesting information from the FORTRAN analysis routines. Answers not available from the data base are requested from the meteorologist running the expert system.

A point forecast is made for a grid point, which coincides with the grid mesh of the LFM model output data. This grid mesh is roughly 200 km on a side. WILLARD is run repeatedly over a 14 × 10 grid mesh covering the central US. The result of this run is an array of 140 point forecasts stored in a disk file. Subjective contour analysis is used to delineate areas with the same density/risk categories (some smoothing of the categories is done for verification purposes). These areas are plotted on a base relief map of the central US.

WILLARD's knowledge base was developed over the course of several months by a meteorologist familiar with severe thunderstorm forecasting procedures. Discussions with former SELS forecasters and others identified main parameters to be examined. The meteorological literature was used to understand better the effects of various relationships between parameters pertaining to severe thunderstorms. Actual weather data during severe, marginally severe, and non-severe weather days were also examined.

It became apparent from reading the meteorological literature and discussing forecasting methods that no coherent system of rules covering all possible severe storm cases had yet been synthesized. The availability of an expert system building facility called RuleMaster (see Michie *et al.*, 1984), which could build classification rules by rule induction, was thought to provide a potential solution. Using this system, classification rules are induced by generalization over examples of expert decision-making. An example is expressed as a vector of values pertaining to attributes of the decision, together with the expert's classification (Michie *et al.*, 1984; see also Quinlan, 1979).

For purposes of rapid development, subjectively selected examples were used to build the prototype expert system. Cases of real weather data were subsequently applied in the ongoing refinement of WILLARD.

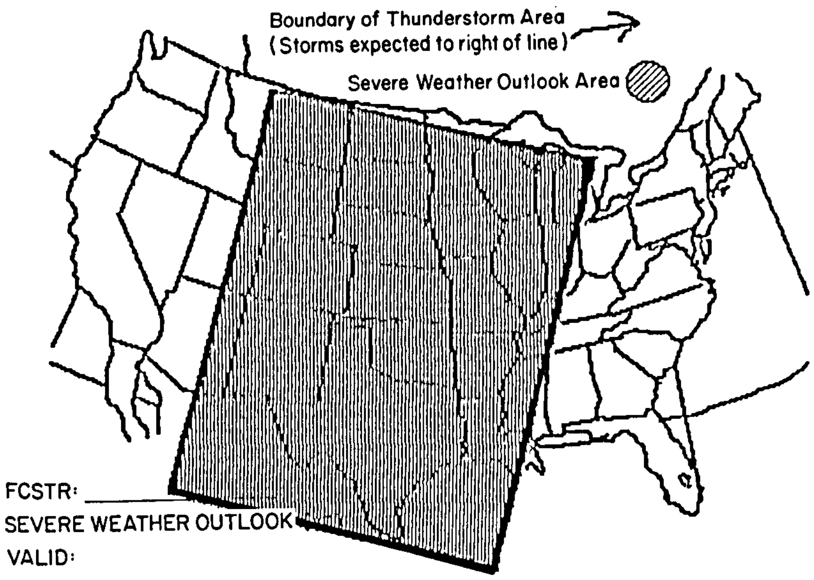

Boundary of Thunderstorm Area
(Storms expected to right of line)
Severe Weather Outlook Area
FCSTR:
SEVERE WEATHER OUTLOOK
VALID:

Figure 1. Map of central United States representing WILLARD's forecast domain area (hatched area).

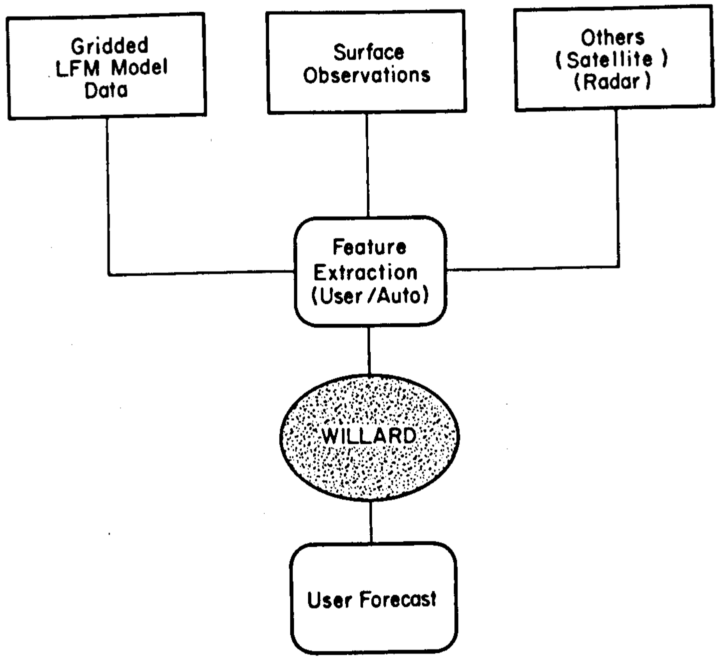


Figure 2. Information flow diagram for WILLARD.

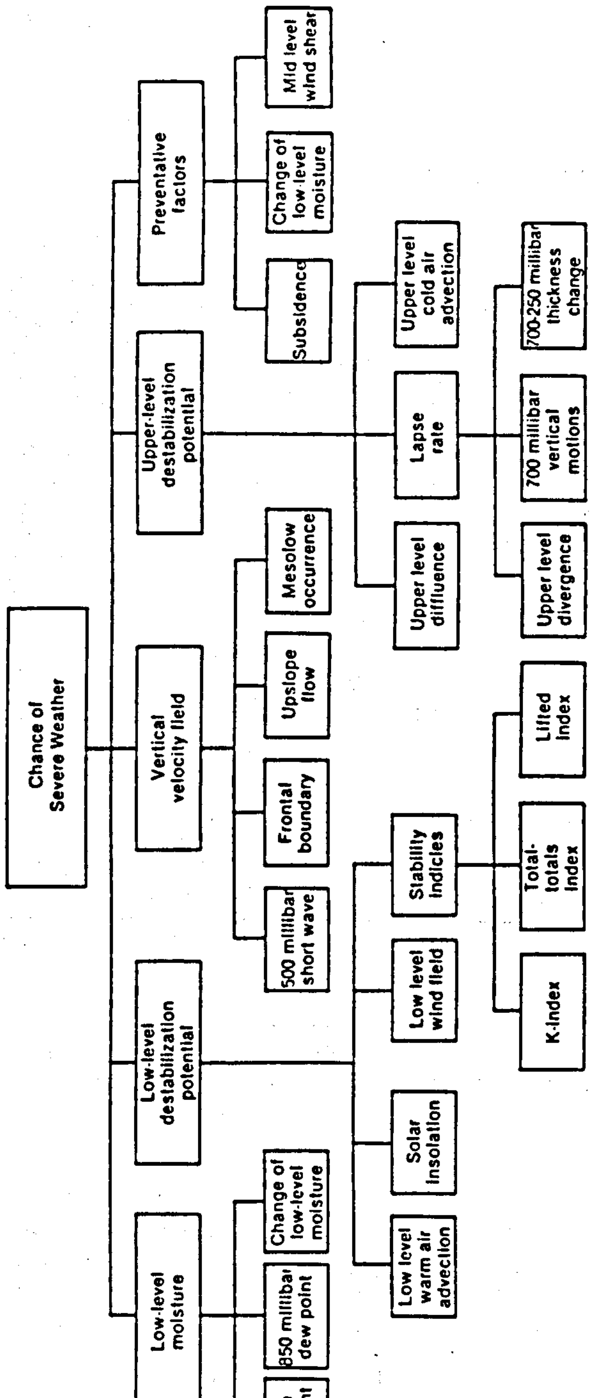


Figure 3. Hierarchical structure of WILLARD's knowledge-base.

The WILLARD expert system is composed of a hierarchy of 30 modules (Figure 3), each containing a single decision rule. This hierarchy is on average four levels deep. All decision rules within each module were developed using inductive generalization (except for some looping control for executing over the grid of data). About 140 examples out of a possible nine million situations were used in building WILLARD.

For the top-level module (Chance of Severe Weather in Figure 3), inductive generalization was able to order the critical meteorological factors in a manner consistent with the way forecasters perform their analysis. For example, if the key factors were all totally unfavourable, a rapid decision could be made: otherwise, more parameters were investigated until a decision could be reached.

WILLARD was designed to operate in either manual or automatic forecast mode. In manual mode, the system asks the meteorologist about pertinent weather conditions for the forecast area and produces a complete, reasoned forecast. In automatic mode, WILLARD obtains necessary information from National Meteorological Center data files (viz., the LFM gridded data), with some information obtained from a meteorologist interactively.

The form of a typical decision rule along with the attribute value set is shown in Figure 4. This rule is used in determining whether the low-level

EXAMPLE SET

| solar insolation | low-level jet | indices | | |
|---|---|---|---|---|
| strong | present | marginal | => | (favor, GOAL) |
| strong | absent | weak | => | (unfav, GOAL) |
| weak | present | strong | => | (margin, GOAL) |
| strong | absent | strong | => | (favor, GOAL) |
| strong | absent | marginal | => | (margin, GOAL) |
| weak | absent | marginal | => | (unfav, GOAL) |
| weak | present | marginal | => | (margin, GOAL) |

INDUCED RULE FROM 'LL_DSTAB.IND'

```
[indices]
        weak : => ( unfav, GOAL )
    marginal : [solar_insol]
                strong : [low_level_jet]
                        present: => ( favor, GOAL )
                         absent : => ( margin, GOAL )
                  weak : [low_level_jet]
                        present: => ( margin, GOAL )
                         absent : => ( unfav, GOAL )
      strong  : [solar_insol]
                 strong: => ( favor, GOAL )
                   weak: => (margin, GOAL )
```

Figure 4. Example of induction set and corresponding decision rule.

destabilization is unfavourable, marginal, or favourable for severe thunderstorm formation.

The decision rules for all of the modules shown in Figure 3 were examined by a meteorologist for correctness and consistency by applying the rules individually and collectively using actual severe weather data. The meteorologist could then change attribute values and induce a new set of decision rules until finally he was satisfied with the rules produced.

Another utility of the RuleMaster system, Radial, was used to execute the complete WILLARD expert system. Initial testing uncovered numerous errors in both the knowledge-base and also in the FORTRAN analysis routines. In mid-1984, after these errors were rectified, it was decided that the WILLARD knowledge-base must remain static while a series of validation runs were completed and analysed. The results of this validation and analysis are presented in the next section. These results should be viewed as the first verification data of a prototype weather forcasting expert system. They have already provided direction for future improvements and refinements in both the WILLARD knowledge-base and analysis package.

## 3. VERIFICATION

The methodology used in verifying WILLARD's and SELS' forecasts was the same as that used by NSSFC researchers (Weiss *et al.*, 1980). Each severe thunderstorm outlook area is examined to produce statistical data useful in evaluating the forecasts. Since WILLARD produces a severe thunderstorm outlook similar to those produced by NSSFC forecasters, this was a reasonable verification method.

Three test case study days are discussed in detail and included as an Appendix. The study days were 29 April 1984, 25 May 1984, and 7 June 1984. Two of these days (29 April and 7 June) were chosen to highlight forecasting abilities during major severe weather outbreaks. The other day (25 May 1984) was chosen as a day representing a minor outbreak day.

Actual SELS outlooks contain forecast reasoning on why an area could experience severe thunderstorms. This reasoning was compared with WILLARD's reasoning for each of the test case study days. Actual weather data for these test days were consulted to aid in interpreting the reasoning behind each forecast. The following discussion of verification statistics follows closely the discussion found in Weiss *et al.* (1980).

### 3.1. Methodology for verification statistics

#### 3.1.1. *Definitions of major verification statistics*

Verification of severe thunderstorm outlooks is based upon the critical skill index (*CSI*) (Donaldson *et al.*, 1975) applied over a large area. It is the ratio of successful predictions of severe weather to the number of severe events that occurred or were forecast to occur. *CSI* scores over 0.5 are considered good by SELS forecasters.

For purposes of severe weather verification, the *CSI* is first computed by dividing all weather events for a given outlook into four groups:

*x*—severe storm reports correctly predicted (i.e. those reports found within a severe risk outlook area);

*y*—severe storm reports not predicted (i.e. those lying outside the severe risk outlook area);

*z*—non-severe weather predicted as severe; and

*w*—non-severe weather correctly predicted.

The probability of detection (*PoD*) is the proportion of severe weather events correctly forecast:

$$PoD = x/(x+y) \tag{1}$$

An outlook area that contains all of the severe weather reports will have a *PoD* of unity. The *PoD* is normally expressed in per cent, so that its range is 0–100 per cent. A *PoD* of 100 per cent is the best value for an outlook.

The false alarm ratio (*FAR*) is the proportion of predictions that fail to verify:

$$FAR = z/(z+x). \tag{2}$$

The *FAR* ranges between 0 and 1. A *FAR* of 0 indicates a perfect forecast. The *FAR* is modified by the use of an areal distribution term (Weiss *et al.*, 1980), which quantitatively determines the amount of over-forecasting from either the outlook area being too large or insufficient density of severe reports. Thus while a high *PoD* is obtained when a large percentage of severe weather events occurs within a forecaster's outlook area, he is discouraged from forecasting excessively large areas which would increase the *FAR* and decrease the *CSI*.

The *CSI* can now be expressed in terms of *PoD* and *FAR*.

$$CSI = x/(x+y+z) = \{(1/PoD) + [1/(1-FAR)] - 1\}^{-1}. \tag{3}$$

The *CSI* ranges from zero to unity, with higher numbers indicating better forecasts. The *CSI* is also known as the Threat score. Some outlooks may forecast severe weather in several unconnected regions. Here, separate *FARs* are calculated for each area, and an area-weighted average is computed for the entire outlook. This average *FAR* is then used with the total percentage of all severe events within the forecast area (*PoD*) to calculate a single *CSI* via equation (3) for the entire outlook for that day (Weiss *et al.*, 1980).

### 3.1.2. *Other statistical parameters*

The extent of areal coverage ($CA$) is that portion of the outlook area in which severe weather events occur. It is defined as:

$$CA = [(\text{no. of MDR blocks with severe events}) \times K]/(\text{outlook area}) \tag{4}$$

where a MDR block can only be counted once no matter how many severe events might be clustered within a single MDR block (by definition at NSSFC). The constant $K$ is equal to 1681 sq km (484 sq sm), the area covered by a single MDR block.

The coverage bias ($CBIAS$) is defined as the ratio of the actual areal coverage to the forecast areal coverage. The forecast areal coverage is determined from the outlook risk category of the outlook. If the actual areal coverage lies within the range of the forecast risk category, then no coverage bias exists (i.e. $CBIAS = 1.0$). If the actual areal coverage is outside the range of the forecast risk category, the forecast coverage is taken as the category extreme closest to the actual coverage.

The good area is that portion of the outlook area affected by severe weather. This statistic incorporates both the forecast areal coverage and the areal distribution of severe reports within the outlook area. In particular, for a Slight risk each event is assumed to affect a $6 \times 6$ MDR grid array surrounding the event (2.77 per cent areal coverage); for a Moderate risk each event affects a $4 \times 4$ array (6.25 per cent areal coverage); and for a High risk each event affects a $3 \times 3$ array (11.11 per cent areal coverage). The total number of MDR blocks determined in this manner is summed to compute the good area. Each MDR block can be counted only once using this method. If the good area is the same as the original outlook area, then this is considered a representative forecast (although, this outlook could still miss severe events outside its area).

A $FAR$ can also be defined as one minus the good area percentage (the proportion of the outlook area affected by severe weather), or

$$FAR = 1 - (\text{affected area/area of ourlook}). \tag{5}$$

The bad area is that portion of the outlook area not affected by severe weather, and is defined as:

$$\text{bad area} = (\text{area of outlook}) - (\text{good area}). \tag{6}$$

A bad area equal to zero would mean that the good area is equal to the original area of the outlook (which is what one desires). The sum of the good and bad areas equals the original outlook area.

## 3.2. Overall verification statistics

This section discusses the verification results for both WILLARD and SELS on the selected days during the spring 1984 central US peak thunderstorm season. There were a total of 30 WILLARD severe thunderstorm outlooks generated spanning a period from 22 April through 11 June 1984. Since the NSSFC verification scheme is only applicable on days when severe weather was outlooked (i.e. a category of Slight, Moderate, or High risk), the actual number of WILLARD outlooks verified by NSSFC was 24, because WILLARD generated six no severe outlooks. Statistics on the 24 WILLARD outlooks verified by NSSFC are given in Table 1.

The overall *PoD* for WILLARD's 24 outlooks was 37 per cent. The *FAR* was 0.628 for these outlooks. These two parameters combined yielded an average *CSI* of about 0.20. Each of these experienced a wide range of daily values. For WILLARD's *PoD*, the range was 0–100 per cent; for the *FAR*, the range was 0.116–1.000; and for the *CSI*, the range was 0.000–0.691.

The average size area for WILLARD outlooks was about 260,000 sq km (100,000 sq sm). This was almost one half the average size for SELS outlooks during this period and a similar springtime period (Weiss and Reap, 1984). The average good area for WILLARD was over 93,000 sq km (36,000 sq sm). The average bad area for these outlooks was about 166,500 sq km (64,300 sq sm). June 7 had the largest good area of all days, with an area of over 114,500 sq sm. June 4 had the largest bad area of 430,399 sq sm. There were two days on which the good area equalled the original outlook area (26 and 29 April) while there were six days on which the bad area equalled the original outlook area (28 April; 2, 22, 23 May; and 1, 2 June).

For the 24 WILLARD outlook forecasts, there were 1001 severe weather reports, 190 of these being tornadoes. WILLARD captured 369 of 1001 reports, including 82 of the tornadoes within its outlook areas. The areal coverage for the test period was 5.5 per cent, for which a forecast Slight risk category would give a coverage bias of unity. WILLARD's category tended to over-forecast slightly as indicated by an average *CBIAS* of 0.830. There were five days when the coverage bias was near unity.

Since WILLARD used fairly large-scale data, its outlooks forecast areas of widespread severe weather rather than isolated severe thunderstorms. When days were chosen that had more than 10 tornadoes, the WILLARD *CSI* became 0.33, with the *PoD* being 40 per cent and the *FAR* being 0.442. In addition, on these days the average WILLARD outlook area became 206,200 sq km (79,610 sq sm). The average good area became 169,600 sq km (65,480 sq sm) with the average bad area being 36,600 sq km (14,130 sq sm). The areal coverage of severe weather on these days was 14.4 per cent, which implies a High risk category being the proper outlook category. The coverage bias on these days was 1.92, implying that WILLARD outlooks tended to under-forecast on these days.

Table 2 contains verification data for SELS 08.00Z convective outlooks for the same 24 days of WILLARD outlooks (P. W. Leftwich, NSSFC,

Table 1. Verification statistics for WILLARD outlooks on selected days in 1984.

| Year | Month | Day | Area | Hits | Severe reports | *PoD* % | Tornado hits | Tornados | % | Areal coverage | *CBIAS* | Far | Good area | Bad area | *CSI* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 84 | 4 | 25 | 44356 | 11 | 19 | 57 | 3 | 6 | 60 | 0.065 | 1.091 | 0.225 | 34364 | 9932 | 0.489 |
| 84 | 4 | 26 | 132775 | 44 | 220 | 20 | 6 | 29 | 20 | 0.102 | 1.580 | 0.409 | 89477 | 43298 | 0.176 |
| 84 | 4 | 27 | 21359 | 10 | 55 | 18 | 2 | 13 | 15 | 0.204 | 3.399 | 0.484 | 21359 | 0 | 0.154 |
| 84 | 4 | 28 | 42001 | 0 | 1 | 0 | 0 | 0 | 0 | 0.000 | 0.000 | 1.000 | 0 | 42001 | 0.000 |
| 84 | 4 | 29 | 93899 | 78 | 142 | 54 | 12 | 26 | 46 | 0.258 | 3.127 | 0.335 | 93899 | 0 | 0.424 |
| 84 | 5 | 2 | 8507 | 0 | 121 | 0 | 0 | 19 | 0 | 0.000 | 0.000 | 1.000 | 0 | 8507 | 0.000 |
| 84 | 5 | 3 | 32900 | 4 | 13 | 30 | 0 | 1 | 0 | 0.059 | 0.981 | 0.655 | 15725 | 17175 | 0.191 |
| 84 | 5 | 22 | 61391 | 0 | 0 | 0 | 0 | 0 | 0 | 0.000 | 0.000 | 1.000 | 0 | 61391 | 0.000 |
| 84 | 5 | 23 | 13507 | 0 | 2 | 0 | 0 | 1 | 0 | 0.000 | 0.000 | 1.000 | 0 | 13507 | 0.000 |
| 84 | 5 | 24 | 91148 | 8 | 25 | 32 | 1 | 4 | 25 | 0.037 | 1.000 | 0.124 | 79860 | 11288 | 0.306 |
| 84 | 5 | 25 | 77742 | 15 | 22 | 68 | 1 | 2 | 50 | 0.068 | 1.000 | 0.477 | 40656 | 37086 | 0.420 |
| 84 | 5 | 26 | 182077 | 10 | 11 | 90 | 0 | 0 | 0 | 0.019 | 0.310 | 0.902 | 17908 | 164169 | 0.097 |
| 84 | 5 | 27 | 80496 | 16 | 60 | 26 | 0 | 2 | 0 | 0.060 | 1.002 | 0.212 | 63404 | 17092 | 0.243 |
| 84 | 5 | 28 | 167512 | 1 | 1 | 100 | 0 | 0 | 0 | 0.003 | 0.029 | 0.974 | 4356 | 163256 | 0.026 |
| 84 | 5 | 31 | 20347 | 1 | 8 | 12 | 1 | 6 | 16 | 0.024 | 1.000 | 0.144 | 17424 | 2923 | 0.119 |
| 84 | 6 | 1 | 25783 | 0 | 12 | 0 | 0 | 0 | 0 | 0.000 | 0.000 | 1.000 | 0 | 25783 | 0.000 |
| 84 | 6 | 2 | 50828 | 0 | 3 | 0 | 0 | 0 | 0 | 0.000 | 0.000 | 1.000 | 0 | 50828 | 0.000 |
| 84 | 6 | 4 | 480251 | 11 | 22 | 50 | 4 | 7 | 57 | 0.009 | 0.151 | 0.896 | 49852 | 430399 | 0.094 |
| 84 | 6 | 5 | 112626 | 7 | 38 | 18 | 2 | 8 | 25 | 0.021 | 0.519 | 0.553 | 50336 | 62290 | 0.147 |
| 84 | 6 | 6 | 166468 | 3 | 23 | 13 | 0 | 1 | 0 | 0.006 | 0.291 | 0.878 | 20328 | 146140 | 0.067 |
| 84 | 6 | 7 | 137934 | 86 | 120 | 71 | 30 | 39 | 76 | 0.186 | 1.430 | 0.309 | 114574 | 23360 | 0.539 |
| 84 | 6 | 9 | 175378 | 13 | 14 | 92 | 2 | 2 | 100 | 0.028 | 0.275 | 0.838 | 28556 | 147322 | 0.160 |
| 84 | 6 | 10 | 104630 | 16 | 23 | 69 | 4 | 6 | 66 | 0.051 | 0.753 | 0.547 | 47432 | 57198 | 0.377 |
| 84 | 6 | 11 | 83185 | 35 | 46 | 76 | 14 | 19 | 73 | 0.116 | 1.939 | 0.116 | 73568 | 9617 | 0.691 |
| FCSTR WILLARD | Number outlooks | | Average area | Hits | Reports | *PoD* | Tornado hits | Tornados | % | Areal coverage | *CBIAS* | *FAR* | Good area | Bad area | *CSI* |
| 49 | 24 | | 100321 | 369 | 1001 | 37 | 32 | 190 | 43 | 0.035 | 0.830 | 0.628 | 35962 | 64339 | 0.197 |

Table 2. Verification of SELS convective outlooks issued at 0800Z on selected days in 1984 (Leftwich, NSSFC, personal communication, 1987)

| Date | Area | *PoD* | *CBIAS* | *FAR* | *Good area* | *Bad area* | *CSI* |
|---|---|---|---|---|---|---|---|
| 4/25 | 126180 | 0.80 | 1.00 | 0.44 | 70664 | 55516 | 0.49 |
| 4/26 | 307095 | 0.89 | 1.85 | 0.44 | 218621 | 88474 | 0.52 |
| 4/27 | 476273 | 0.77 | 1.04 | 0.53 | 225060 | 251213 | 0.41 |
| 4/28 | 65578 | 0.00 | 0.00 | 1.00 | 0 | 66578 | 0.00 |
| 4/29 | 366116 | 0.83 | 1.00 | 0.49 | 185856 | 180260 | 0.46 |
| 5/2 | 366354 | 0.66 | 1.34 | 0.50 | 182468 | 183886 | 0.40 |
| 5/3 | 399755 | 0.97 | 1.00 | 0.36 | 257972 | 141783 | 0.63 |
| 5/22 | 261119 | 0.66 | 0.19 | 0.91 | 23232 | 237887 | 0.09 |
| 5/23 | 135950 | 0.55 | 1.00 | 0.46 | 73568 | 62382 | 0.38 |
| 5/24 | 303966 | 0.82 | 1.00 | 0.50 | 152944 | 151022 | 0.45 |
| 5/25 | 214510 | 0.93 | 1.00 | 0.52 | 103092 | 111418 | 0.46 |
| 5/26 | 181739 | 0.16 | 0.40 | 0.76 | 43076 | 138663 | 0.11 |
| 5/27 | 214855 | 0.90 | 1.28 | 0.30 | 150524 | 64331 | 0.65 |
| 5/28 | 94522 | 0.00 | 0.00 | 1.00 | 0 | 94522 | 0.00 |
| 5/31 | No severe forecast | | | | | | |
| 6/1 | 69430 | 0.41 | 1.00 | 0.33 | 46464 | 22966 | 0.34 |
| 6/2 | 67122 | 0.16 | 0.36 | 0.74 | 17424 | 49698 | 0.11 |
| 6/4 | 245426 | 0.63 | 1.00 | 0.55 | 110352 | 135074 | 0.36 |
| 6/5 | 387216 | 0.81 | 1.00 | 0.56 | 169884 | 217332 | 0.40 |
| 6/6 | 493657 | 0.73 | 0.59 | 0.79 | 105028 | 388629 | 0.20 |
| 6/7 | 413460 | 0.94 | 1.45 | 0.47 | 241457 | 172003 | 0.51 |
| 6/9 | 323633 | 0.68 | 0.67 | 0.71 | 92928 | 230705 | 0.25 |
| 6/10 | 413827 | 0.87 | 0.67 | 0.64 | 147136 | 266691 | 0.34 |
| 6/11 | 284051 | 0.84 | 1.00 | 0.57 | 123420 | 160631 | 0.40 |
| Number outlooks | Average area | *PoD* | | *FAR* | Average good area | Average bad area | *CSI* |
| 24 | 270080 | 0.81 | — | 0.56 | 119181 | 150942 | 0.40 |

personal communication, 1987). During this period, the *PoD* for SELS was 81 per cent. The *FAR* was 0.56. These two parameters combined yielded a *CSI* of 0.40—which is double WILLARD's *CSI*. SELS' outlooks overall were better than WILLARD's for this period. Figure 5 gives a daily breakdown comparing SELS' and WILLARD's outlooks. Of note is that WILLARD's *CSI* scores tended to be relatively in phase with the trend of SELS' *CSI* scores.

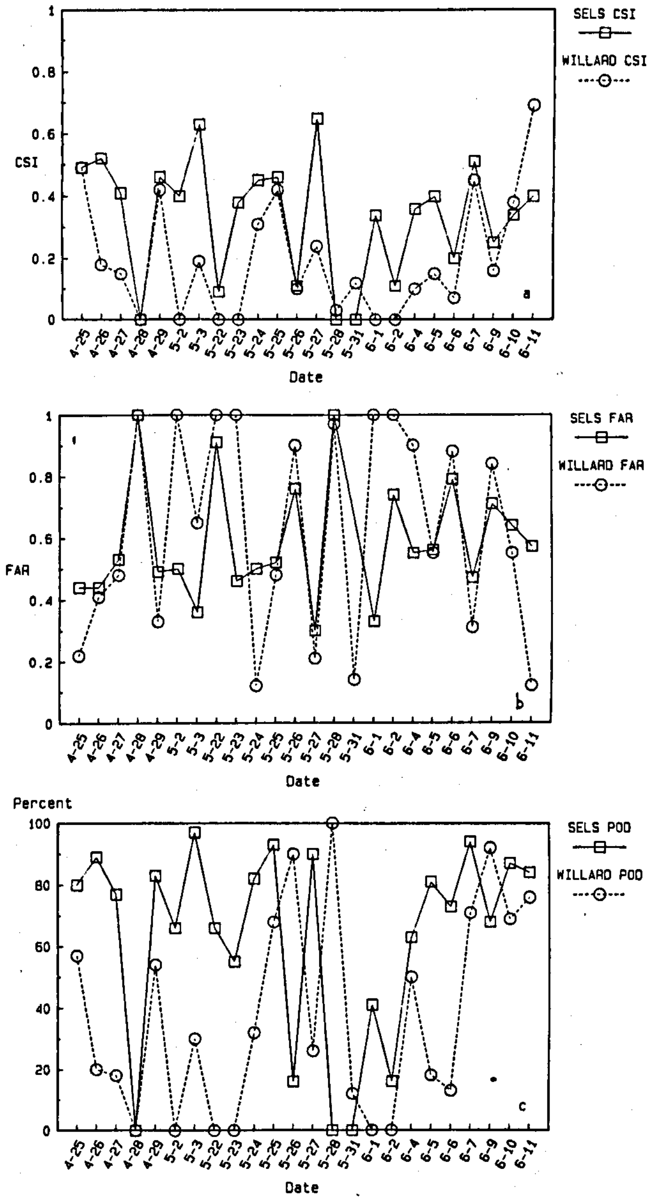


Figure 5. Comparative verification statistics for SELS and WILLARD 08Z Convective Outlooks for Selected Days in Spring 1984: a) *CSI*, b) *FAR*, c) *PoD*.

Examining the six days (viz., 22, 23, 24 April; 1, 29, 30 May 1984) when no severe thunderstorms were forecast by WILLARD, it was found that SELS forecast no severe thunderstorms on those same days with one exception. On 1 May 1984 SELS issued a Slight risk area that covered northeastern Texas, southeastern Oklahoma and western Louisiana. There were two dozen severe weather reports within SELS' outlook area 1 May. The reason why WILLARD did not forecast severe weather on this day was because of insufficient moisture in the lower levels of the LFM model data. Examining the forecast reasoning for SELS and WILLARD on the other non-severe outlook days showed that both agreed that lack of moisture was the primary cause for the non-severe outlook forecast. A detailed analysis of the reasoning given by WILLARD and SELS for the test case study days is given in the Appendix.

## 4. SUMMARY AND CONCLUSIONS

This paper examined two sources of advice for severe thunderstorm forecasting: an outlook produced by a government agency (SELS/NSSFC) and an outlook given by an expert system (WILLARD). Overall the two sets of advice were comparable in critical skill index, and forecast reasoning but different in several other statistical parameters. The WILLARD forecast is comparable to the actual SELS forecast in overall skill on major and most minor outbreak days.

The valid time period for the 08.00 SELS outlook covers a 24-h period whereas the 08.00Z WILLARD outlook spans a 12-h period. This could affect the results presented in this paper. The forecasting strategy behind WILLARD's production of a 08.00Z outlook is to use only the 24-h LFM Model forecast from the 00.00Z LFM model run to prepare the outlook. This was mainly done because the 36-h gridded LFM model forecast data were unavailable. However, it was felt that ±6 h from the time of the 24-h forecast was valid (viz., 00.00Z the next day) and consistent with the fact that severe thunderstorms generally reach their peak in number and intensity within a few hours of 00.00Z. Examination of the NSSFC severe storm log showed this to be true in most cases used in this study.

Therefore, it is estimated that, although some verification values might change for WILLARD's outlook if a full 24 hours' worth of severe weather reports were used, the change should be insignificant. Conversely, the statisitics for the SELS group should also not differ greatly.

On 2 May 1984, WILLARD's 08.00Z outlook forecast relatively little severe weather to occur anywhere within the forecast domain. However, this day turned out to be a major severe weather outbreak day (NOAA, 1984). There were over 120 severe weather reports in northern Texas, Oklahoma, Arkansas, Louisiana, and western Mississippi including 19 confirmed tornadoes. A script of the original WILLARD run was carefully examined, as were many weather maps and data. The SELS 08.00Z outlook had a Moderate risk area centred over the affected area mentioned above.

After examining relevant data and forecasts for 2 May, it appeared that WILLARD had difficulty in properly classifying the low-level moisture field. The LFM model data near the affected area indicated somewhat drier conditions at lower levels than those actual data showed. WILLARD concluded that low-level moisture was insufficient to support severe thunderstorms in this area. This contradicts data taken from vertical measurements of moisture near the threat area. Since WILLARD's moisture decision rule only examined a few vertical points from the LFM forecast data and did not examine actual sounding data, the effects of a slightly drier air mass as forecast by the LFM model were significant. This could be corrected by inserting more knowledge into the moisture module.

Additionally, on 2 May the LFM model was unable to handle the large number of short-wave, low-pressure troughs—as noted by meteorologists responsible for interpreting satellite imagery at the NWS Satellite Field Services Station located at NSSFC. A satellite interpretation message received near 00.00Z on 3 May 1984 indicated model guidance from earlier in the day was not resolving smaller scale features which were causing most of the severe activity. Examination of the LFM data confirmed this, showing that the model had lumped everything together into an a ill-defined low-pressure trough. Therefore identification of trigger mechanisms was clouded by the unrealistic model output. In the future, it might be possible to develop rules for predicting this condition and add them to the vertical velocity field module of WILLARD.

When there are errors in the LFM forecast data, it is likely that SELS forecasters are able to adjust the data to compensate. WILLARD did not apply any data adjustment of the LFM data nor did it attempt to recognize model errors (it did check for gross data range errors). Installation of rules for adjusting erroneous model forecast data was beyond the scope of this project. It is something which needs to be implemented in future weather forecasting expert systems which automatically provide guidance from numerical models. The rules governing these adjustments are complex and based on pattern recognition. However, an easily expandable system like WILLARD could accommodate addition of these rules.

Overall, the results of this paper are encouraging for pursuing the application of expert system technology to weather forecasting. Further research is underway to improve the knowledge-base to better handle smaller scale severe thunderstorm outbreaks.

**Acknowledgements**

Support for this study was provided by Radian Corporation and is gratefully appreciated. This study benefited from the assistance provided by Dr Preston Leftwich, of the National

Severe Storms Forecast Center, Techniques Development Unit, who graciously provided the necessary verification data on WILLARD and SELS. Drs Charles Doswell and Robert Maddox of the Environmental Research Laboratories, NOAA, supplied many stimulating ideas for this project. Steven Muggleton of the University of Edinburgh supplied much useful information in the structuring and explanation capability of WILLARD. Finally, Charlie Riese of Radian motivated the author to explore the application of expert systems technology to weather forecasting.

## APPENDIX

### 29 April 1984—test case no. 1

This was a major and widespread severe thunderstorm outbreak day. Within the area and time of WILLARD's forecast domain there were 142 severe weather reports including 26 reports of tornadoes. One of these tornadoes caused one death and extensive property damage to the town of Mannford, in northeast Oklahoma (Ferguson *et al.*, 1985). Most of the severe weather reports came from a six state area covering all of Missouri, Iowa, and Illinois, and portions of eastern Oklahoma, Kansas, and Texas. Scattered reports were received from northern portions of Arkansas, Louisiana, and Mississippi, and southern Wisconsin.

The SELS 08.00Z convective outlook issued on 29 April 1984 recognized that meteorological conditions were favourable for a widespread severe weather outbreak in the southern Great Plains States (Oklahoma, Arizona, Missouri, and Kansas). They issued a High risk outlook area that covered most of Oklahoma, Missouri, Illinois, and Arkansas, portions of northern Texas, extreme northwest Louisiana, southern Iowa, western Tennesse, and western Indiana (Figure 6a).

High risk outlooks are rarely issued by SELS forecasters, and are only issued when the threat is clearly recognized as being substantial in severity and areal extent. Although there are usually less than 10 High risk outlooks issued annually by SELS forecasters, their greatest forecasting ability is exhibited when the threat of severe thunderstorms and tornadoes is the highest (Weiss and Reap, 1984). SELS' outlook area on this day covered over 360,000 sq sm.

WILLARD forecast a smaller outlook area evenly divided between Slight and Moderate risk areas covering northeastern Oklahoma, eastern Kansas, and southern two-thirds of Illinois, all of Missouri, and southeast Iowa (Figure 6b). This outlook area covered almost 94,000 sq sm.

In comparing statistical results from the two forecasts, the *CSI* was 0.49 for WILLARD and 0.46 for SELS, which are similar. The *PoD* was 54 per cent for WILLARD contrasted with 83 per cent for SELS. WILLARD had 78 severe reports within its outlook area out of 142, with 12 out of 26 tornadoes included. However, mostly due to the smaller size of WILLARD's outlook, the *FAR* was 0.34 for WILLARD vs. 0.49 for SELS. So even though the *CSI* was similar for both outlooks, the SELS outlook captured a majority of severe weather reported at the expense of an

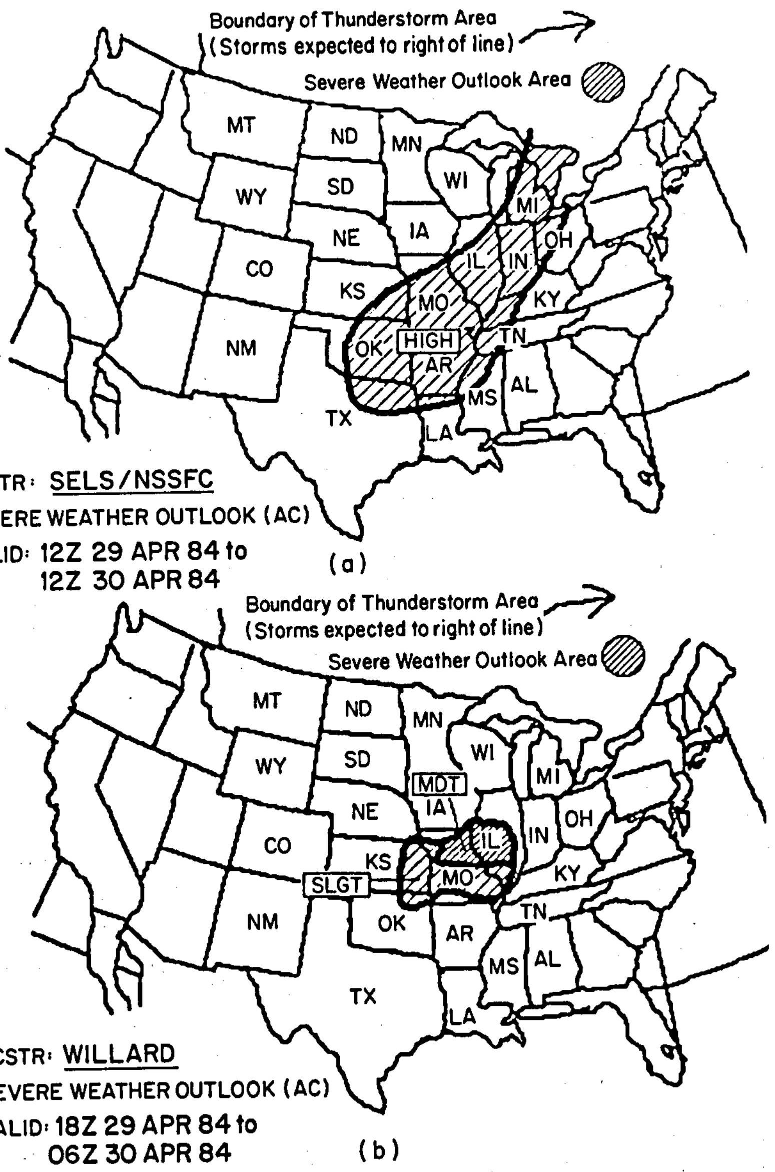

Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area ▨
MT
ND
MN
WI
SD
WY
MI
NE
IA
OH
IL
IN
CO
KS
MO
KY
HIGH
TN
OK
AR
NM
AL
MS
TX
LA
FCSTR: SELS/NSSFC
SEVERE WEATHER OUTLOOK (AC)
VALID: 12Z 29 APR 84 to 12Z 30 APR 84
(a)
Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area ▨
MT
ND
MN
WI
SD
WY
MDT
MI
IA
NE
OH
IL
IN
CO
KS
MO
KY
SLGT
TN
OK
AR
NM
MS
AL
TX
LA
FCSTR: WILLARD
SEVERE WEATHER OUTLOOK (AC)
VALID: 18Z 29 APR 84 to 06Z 30 APR 84
(b)

Figure 6. Map of SELS and WILLARD outlooks—29 April 1984.

increased false alarm ratio. This is reflected in the similarity of the *CSI* scores: a low *PoD* coupled with a low *FAR* can yield a *CSI* score nearly equivalent to a high *PoD* coupled with a high *FAR*. It is a matter of individual preference as to whether *PoD* or *FAR* is of greater importance, and is not an issue in this paper.

The good area for WILLARD's outlook equalled the original outlook area. The areal coverage of severe weather within WILLARD's outlook was almost 26 per cent, with a *CBIAS* of 3.13. This was the highest coverage bias encountered in this study. It indicates that WILLARD under-forecast the severe weather that occurred within its outlook. If WILLARD had forecast a High risk category it would have had a coverage bias of unity. The SELS coverage bias was unity, as they did forecast a High risk category.

In examining the meteorological reasoning behind each forecast it was apparent that both forecasts identified that the interaction of a strong upper level low pressure system with a very warm and moist unstable air mass near the surface, and a source of upper-level venting (or diffluence) would all phase in over eastern Oklahoma and Kansas, moving into Arkansas, Missouri, and later Illinois.

In determining the moisture field, WILLARD found high LFM forecast values of low-level moisture convergence over eastern Oklahoma and Kansas, and all of Missouri and the southern two-thirds of Illinois to occur at 00.00Z, 30 April 1984—the middle of WILLARD's valid time period. Examination of model forecast dew points at an average altitude of 1500 m (850 millibar pressure surface) and near the surface also confirmed this to WILLARD. Further, various stability indices were examined by WILLARD and found to be favourable for severe weather, especially because a triggering mechanism was present.

The SELS forecast reasoning stated that due to the strengthening of the low level wind field during the day (29th), there would be an attendant strong influx of low-level moisture into the threat region. High model forecast values of near-surface dew points were also mentioned as being within the threat area. Both forecast reasonings examined the low-level wind field for location of the maximum wind speeds and found favourable conditions for the formation of severe thunderstorms.

The trigger mechanism was the result of the strong upper-level, low-pressure system, as identified by the large values of vorticity advection approaching both threat areas from the southwest. WILLARD used maximum vorticity analysis from the LFM model data to locate intense activity. This is basically what SELS did, in that the SELS forecaster examined contour maps of LFM forecast vorticity and upper-level wind speed and identified areas experiencing maximum vorticity advection and strong upper-level winds.

While not explicitly mentioned by SELS on their 08.00Z forecast discussion, the presence of a warm frontal boundary lying across southern Missouri and Illinois was picked up by WILLARD as being an important triggering mechanism. It has been found (Maddox *et al.*, 1980) that severe storms often reach maximum intensity within the environment attending sharp thermal boundaries, as was the case on the 29 April 1984.

WILLARD checked for the presence of an upper-level venting mechanism at several grid points and found that there was such a mechanism present in the form of a core of high-speed winds in the upper levels of the atmosphere. This was clearly stated by SELS in their reasoning as being a major contributing factor.

A factor considered by SELS but not included in the WILLARD knowledge-base is the presence of a mid-level dry slot (punch) moving into the threat area. Investigations of the literature show this to be of the prime importance in defining the extent and severity of thunderstorms (Miller, 1972). This may account for part of WILLARD's under-forecasting on this day.

Overall, the two forecasts had similar *CSI*s, but were different in *PoD*, *CBIAS*, and *FAR*. The SELS forecast discussion of their lines of reasoning was generally similar to the explanation given by WILLARD, although the SELS reasoning contained much finer scale details that WILLARD did not have. But when it is considered that this was a widespread severe weather day, most parameters were easily identified from the model forecast data. Additionally, comparison of the LFM forecast maps with later maps that show how conditions actually turned out indicate that the LFM model had a reasonable handle on the major features contributing to severe weather on this day.

### 25 May 1984—test case no. 2

On this day, a slow moving cold front touched off severe thunderstorms along its leading boundary as it moved southeastward across the southern Great Plains region during the afternoon. Because severe thunderstorms formed in a narrow line along the front, severe weather reports were mainly restricted to this narrow frontal zone, and consequently were not very widespread. There were only 22 severe weather reports within the forecast domain consisting of mostly large hail and high winds, with only a few minor tornadoes being observed (NOAA, 1984). Missouri and Illinois were the only states to report severe weather—all of it occurring in thunderstorms along the cold front. This day would be considered a minor severe thunderstorm day by meteorologists.

Even though there were a relatively small number of severe reports, the meteorological conditions favourable for severe weather along the front were present and easily recognized by both SELS and WILLARD (Figure 7). The SELS outlook had a Slight risk threat area that covered northeastern Oklahoma, eastern Kansas, most of Missouri, most of Illinois, and portions of extreme southeast Iowa and northwest Arkansas. WILLARD had a moderate risk outlook for virtually the same area that SELS outlooked, with a narrower major axis than SELS' area. The forecast area for SELS was about 214,000 sq sm, while for WILLARD the area was about 78,000 sq sm.

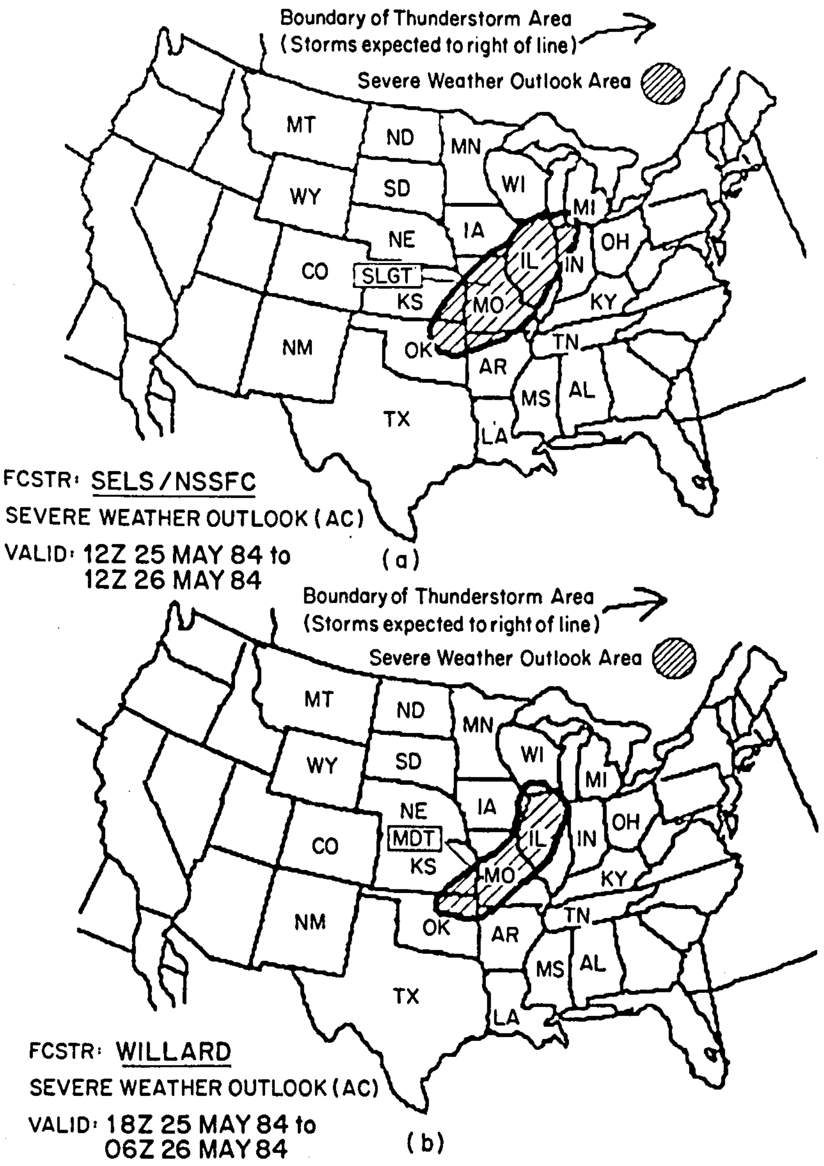

Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area
MT
ND
MN
WI
MI
WY
SD
NE
IA
IL
IN
OH
CO
SLGT
KS
MO
KY
NM
OK
TN
AR
AL
MS
TX
LA
FCSTR: SELS/NSSFC
SEVERE WEATHER OUTLOOK (AC)
VALID: 12Z 25 MAY 84 to 12Z 26 MAY 84
(a)
Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area
MDT
FCSTR: WILLARD
SEVERE WEATHER OUTLOOK (AC)
VALID: 18Z 25 MAY 84 to 06Z 26 MAY 84
(b)

Figure 7. Map of SELS and WILLARD outlooks—25 May 1984.

In terms of the statistical measures of forecast ability, the *CSI* for each forecast were similar. Again, the probability of detection for the SELS outlook was high; 93 per cent in this case, but the *FAR* was also high; being 0.52, resulting in a *CSI* for SELS of 0.46. In contrast, WILLARD had a relatively low *FAR* of 0.48, coupled with a relatively low probability of detection of 68 per cent, to yield a *CSI* of 0.42. So because of the slightly smaller area forecast by WILLARD the *CSI* scores were very close. WILLARD had 15 out of 22 severe reports within its outlook, with one out of two tornadoes also included.

The coverage bias was unity for WILLARD, since the areal coverage was 6.8 per cent. The coverage bias for SELS' outlook slightly under-forecast this day. The good area for WILLARD was one half of the forecast outlook area; almost 41,000 sq sm, while the bad area was about 37,000 sq sm.

Comparing the two forecasts, there was little difference in forecast reasoning behind the two outlook areas: a cold front, in concert with an approaching short wave of low pressure, would provide the primary lifting mechanism of a moist and unstable air mass over Missouri and Illinois, with a strong flow of upper-level winds providing a suitable venting mechanism.

Both forecasts pointed out the strong focusing of low-level moisture convergence along the frontal boundary. In examining the LFM model output, it appeared that the model was a little slow in moving the front southeastward, which may have affected WILLARD's *PoD* because it did not extend the outlook area further into southern sections of Missouri and Illinois (where there were some severe reports).

Both SELS and WILLARD acknowledged the movement of the cold front across the area accompanied by a strong low-level wind field. WILLARD also indicated that if surface heating occurred, any thunderstorms occurring could be severe. Examination of surface data indicated periods of sunshine in the vicinity of the severe weather reports in Missouri and Illinois. WILLARD found the absence of preventive factors was also favourable for severe thunderstorms.

### 7 June 1984—test case no. 3

During an 11-h period from mid afternoon on 7 June to the early morning of 8 June, a massive severe weather outbreak struck many states in the central US unleashing a violent torrent of killer tornadoes covering four states. There were over 120 severe weather reports within WILLARD's forecast domain (NOAA, 1984).

There were 39 confirmed tornadoes occurring primarily in Iowa and Wisconsin, with some tornadoes reported in northern Missouri and southern Minnesota (Figure 8). One of these tornadoes tracked on the ground for a incredible distance of 204 km (127 miles) from northern Missouri across east central Iowa. This was the longest tornado track observed in 1984. It was rated as a devastating tornado for most of its existence and was responsible for three deaths and extensive property and agricultural damage (NOAA, 1984). It completely levelled or extensively damaged all buildings in the towns of Wright and Delta, Iowa resulting in property damage exceeding 30 million dollars.

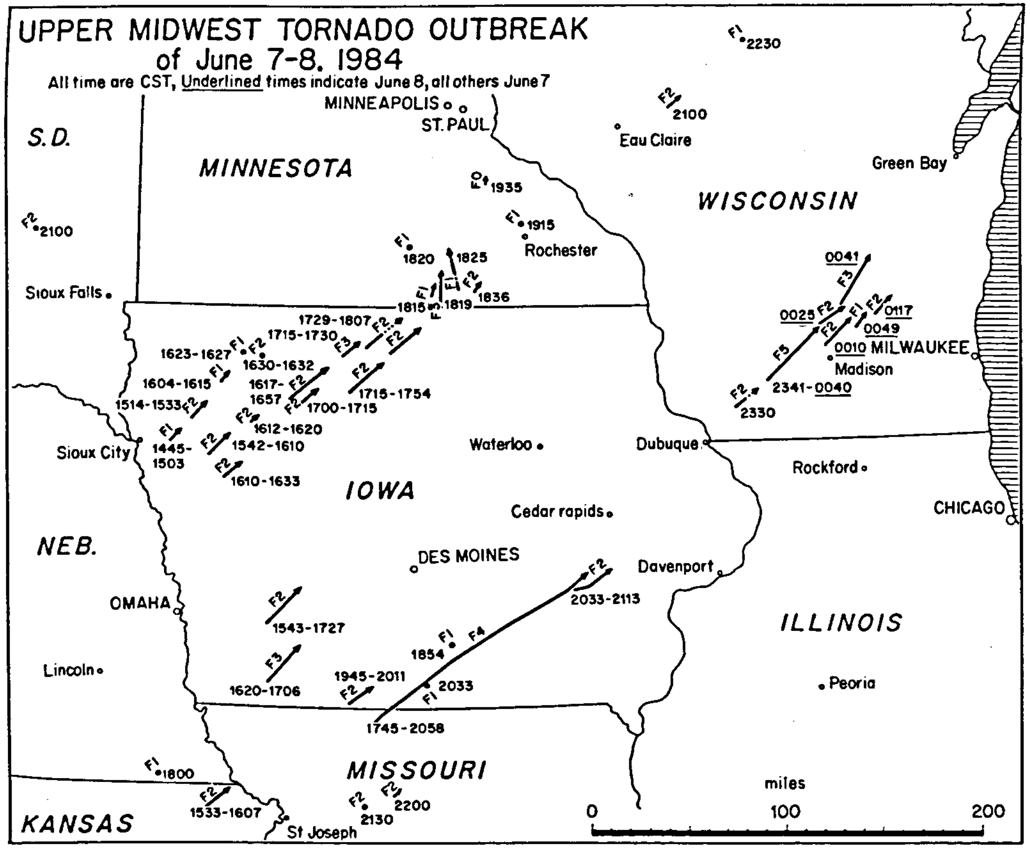


Figure 8. Upper midwest tornado outbreak of 7–8 June 1984 (from Storm Data—NOAA, 1984).

On this day the most violent tornado of the 1984 season demolished the town of Barneveld in south central Wisconsin and killed nine people. It was the most powerful tornado of its class since 1982 and only the second since 1977 to reach such great intensity (Ferguson *et al.*, 1985). Total property damage from this storm that tracked over 58 km (36 miles) on the ground exceeded 40 million dollars. Winds were estimated in excess of 420 km $h^{-1}$ (261 miles $h^{-1}$) near Barneveld.

The SELS 08.00Z outlook called for a Moderate risk of severe thunderstorms over portions of northeastern Kansas, northwestern Missouri, the western two-thirds of Iowa, extreme western Wisconsin, southern Minnesota, southeastern South Dakota, and eastern Nebraska (Figure 9a). This Moderate risk area was enclosed within a large Slight risk area as shown in Figure 9a, which included large portions of Wisconsin, Minnesota, and Kansas. The Moderate risk area covered about 190,000 sq sm, while the entire outlook spanned roughly 413,000 sq sm. The SELS forecasters noted that the stage appeared set for a more active day than the past few days. All of the killer tornadoes that occurred in Iowa were contained in SELS' Moderate risk outlook, while the tornado that struck Barneveld was under a Slight risk.

WILLARD had a complex outlook, with two Slight risk areas flanking a fairly large High risk area (Figure 9b). The High risk area covered most of the Iowa and southern Minnesota, and the northwest portion of Missouri. All of the killer tornadoes that occurred in Iowa were contained in WILLARD's High risk outlook area. The western Slight risk area covered central and northeast Kansas, south central and southeast Nebraska, and portions of extreme northwest Missouri and southwest Iowa. The eastern Slight risk area from WILLARD covered northeastern Missouri, central Illinois, and southeastern Iowa. Notice that no severe weather was outlooked by WILLARD for south central Wisconsin, where the Barneveld killer tornado occurred. The total WILLARD outlook area encompassed almost 138,000 sq sm, with the High risk area covering about 80,000 sq sm.

The verification statistics show that, again, the *CSI* for each forecast was similar; SELS had a *CSI* of 0.51 while WILLARD's *CSI* was 0.54; a forecast is considered a 'good' one by members of SELS if the *CSI* is above 0.5 (Leftwich, NSSFC, personal communication, 1985). The same trend discussed in the previous test study day (no. 2) continued here. The probability of detection for the SELS outlook was 94 per cent for all severe reports and 87 per cent for all reports of tornadoes. WILLARD's outlook had a *PoD* of 71 per cent for all severe weather reports and 76 per cent for all reports of tornadoes.

The *FAR* was again lower for WILLARD, with a value of 0.31 compared to SELS' *FAR* value of 0.47. In this case, the primary threat areas denoted by SELS as a Moderate risk area and by WILLARD as a High risk area were very similar in areal extent and region, as an examination of Figure 9 shows. However, the Slight areas were different: SELS had more areal coverage of Slight risk than did WILLARD, and this might have accounted for the slightly higher *FAR* for SELS.

Examination of forecast reasoning given in the 08.00Z SELS outlook and the explanation given by WILLARD show that most of the major

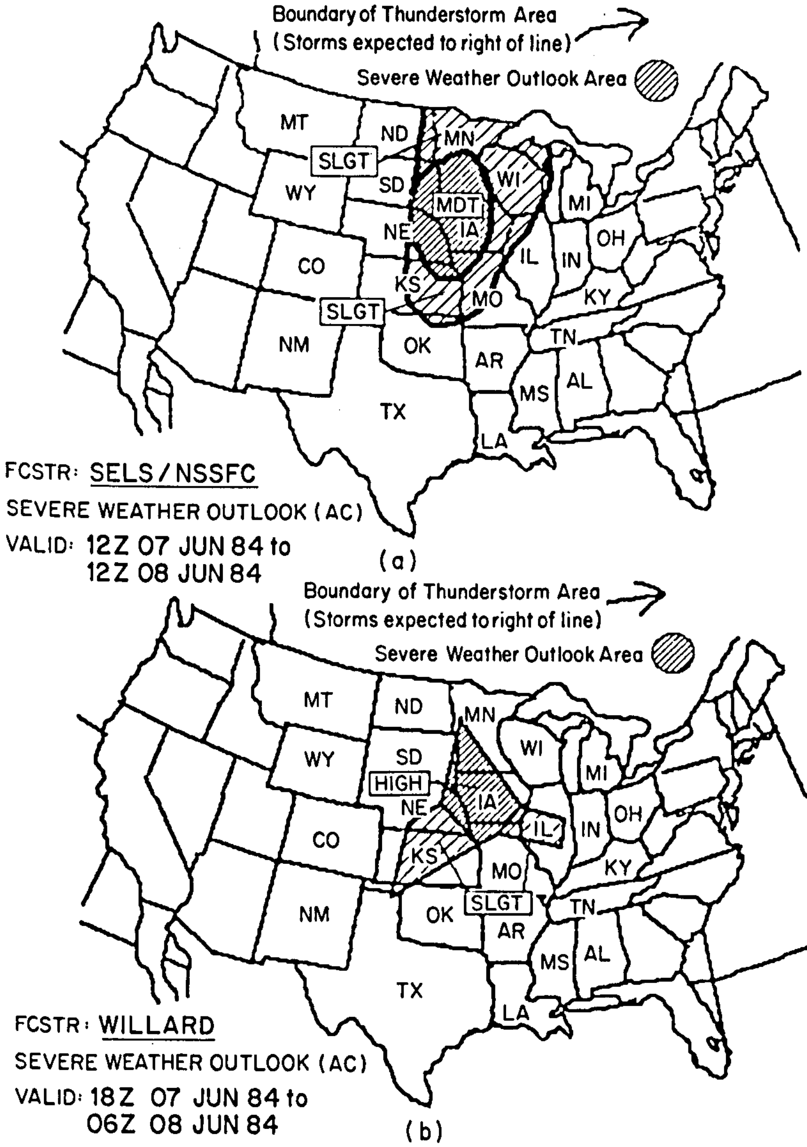
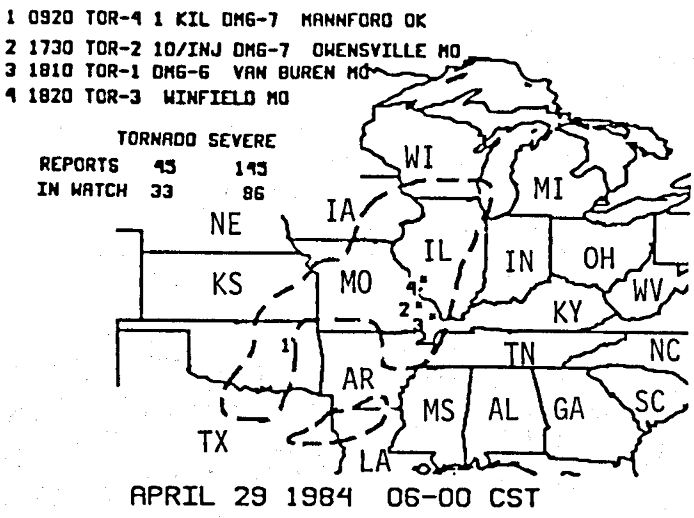

Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area ▨
MT
ND
MN
SLGT
WY
SD
WI
MDT
MI
NE
IA
OH
CO
IL
IN
KS
MO
KY
SLGT
TN
NM
OK
AR
AL
MS
TX
LA
FCSTR: SELS/NSSFC
SEVERE WEATHER OUTLOOK (AC)
VALID: 12Z 07 JUN 84 to 12Z 08 JUN 84
(a)
Boundary of Thunderstorm Area (Storms expected to right of line)
Severe Weather Outlook Area ▨
MT
ND
MN
WI
WY
SD
HIGH
MI
NE
IA
OH
IL
IN
CO
KS
MO
KY
SLGT
TN
NM
OK
AR
AL
MS
TX
LA
FCSTR: WILLARD
SEVERE WEATHER OUTLOOK (AC)
VALID: 18Z 07 JUN 84 to 06Z 08 JUN 84
(b)

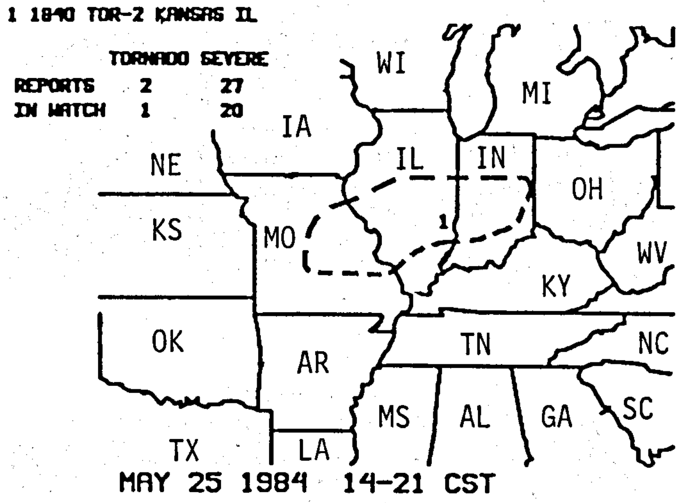
Figure 9. Map of SELS and WILLARD outlooks—7 June 1984.



```
1 0920 TOR-4 1 KIL DMG-7  MANNFORD OK
2 1730 TOR-2 10/INJ DMG-7  OWENSVILLE MO
3 1810 TOR-1 DMG-6  VAN BUREN MO
4 1820 TOR-3  WINFIELD MO

           TORNADO SEVERE
REPORTS      45      145
IN WATCH     33       86
```

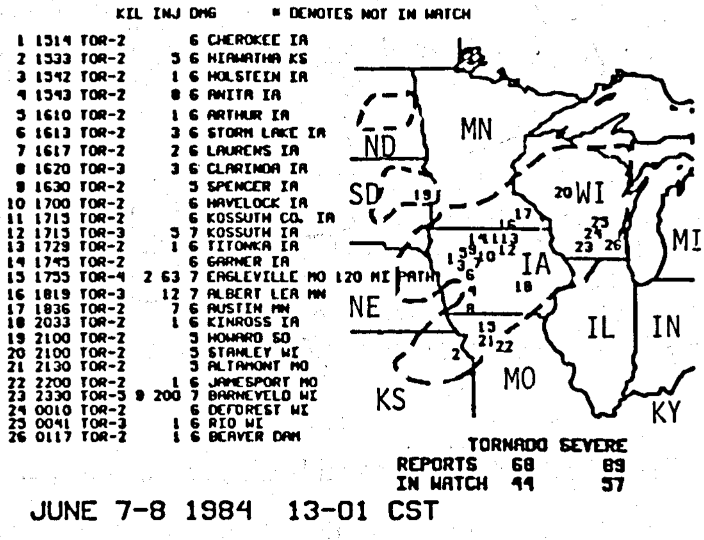
APRIL 29 1984  06-00 CST



```
1 1840 TOR-2 KANSAS IL

           TORNADO SEVERE
REPORTS      2       27
IN WATCH     1       20
```

MAY 25 1984  14-21 CST



```
                    KIL INJ DMG       * DENOTES NOT IN WATCH
 1 1314 TOR-2             6 CHEROKEE IA
 2 1333 TOR-2           3 6 HIAWATHA KS
 3 1342 TOR-2           1 6 HOLSTEIN IA
 4 1343 TOR-2           8 6 ANITA IA
 5 1610 TOR-2           1 6 ARTHUR IA
 6 1613 TOR-2           3 6 STORM LAKE IA
 7 1617 TOR-2           2 6 LAURENS IA
 8 1620 TOR-3           3 6 CLARINDA IA
 9 1630 TOR-2             5 SPENCER IA
10 1700 TOR-2             6 HAVELOCK IA
11 1713 TOR-2             6 KOSSUTH CO. IA
12 1713 TOR-3           3 7 KOSSUTH IA
13 1729 TOR-2           1 6 TITONKA IA
14 1745 TOR-2             6 GARNER IA
15 1755 TOR-4       2  63 7 EAGLEVILLE MO 120 MI PATH
16 1819 TOR-3          12 7 ALBERT LEA MN
17 1836 TOR-2           7 6 AUSTIN MN
18 2033 TOR-2           1 6 KINROSS IA
19 2100 TOR-2             5 HOWARD SD
20 2100 TOR-2             5 STANLEY WI
21 2130 TOR-2             5 ALTAMONT MO
22 2200 TOR-2           1 6 JAMESPORT MO
23 2330 TOR-5       9 200 7 BARNEVELD WI
24 0010 TOR-2             6 DEFOREST WI
25 0041 TOR-3           1 6 RIO WI
26 0117 TOR-2           1 6 BEAVER DAM

           TORNADO SEVERE
REPORTS      68      89
IN WATCH     44      57
```

JUNE 7-8 1984  13-01 CST

Figure 10. Significant severe weather events for the three days discussed in Appendix (dashed line encloses area of days' organized severe activity), modified from Hales and Crowther (1985).

factors were identified in each. Both determined that a significant upper-level trigger mechanism was present in the form of short-wave, low-pressure trough moving out of Nebraska into southern Minnesota during the period. Both found areas of strong low-level moisture convergence across Iowa and southern Minnesota. Both identified a suitable upper-level venting mechanism occurring over the region.

There was mention in the SELS forecast reasoning of a 'dry punch' (a narrow tongue of dry air at mid levels) extending from northwest Missouri into southern Minnesota and southwestern Wisconsin. Examination of Figure 9 shows that this feature was possibly a key parameter to defining the general direction and movement of the tornadoes (Miller, 1972). WILLARD did not perform any checks for this type of feature.

The coverage bias for WILLARD's outlook was 1.5 compared with SELS' outlook bias of 1.5. Both of these values indicate that each outlook under-forecast the actual areal extent of severe weather, which was about 19 per cent of the MDR blocks. However, WILLARD did appear to have the proper category in the major Iowa tornado outbreak area.

Overall, both forecasts were similar in the critical score index, category, and areal coverage, but different in *PoD* and *FAR*. The WILLARD forecast strategy appeared consistent with the relevant features indicated in the 08.00Z SELS outlook discussion. An examination of the LFM model output with model verification data showed the 24-h model forecast used by both SELS and WILLARD to be fairly reasonable in handling the major feature associated with this severe outbreak day. Therefore, since the LFM model input data into WILLARD was reasonable, it appears that the procedural rules used by WILLARD in producing the outlook areas were comparable to SELS' line(s) of reasoning.

## REFERENCES

Crisp, C. A. (1979) Training guide for severe weather forecasters. *Air Weather Service, Tech. Note 79/002*. (Available from NTIS, Springfield, Va., No. ADA083125).

Donaldson, R. J., Dyer, R. M., and Kraus, M. J. (1975) An objective evaluator of techniques for predicting severe weather events. *Preprints 9th Conf. on Severe Local Storms,* Norman, OK, American Meteorological Society, pp. 321–6,

Ferguson, E. W., Otsby, F. P., and Leftwich, P. W., Jr (1985) Tornadoes cause record number of fatalities. *Weatherwise* **38,** 20–25, 48–49.

Foster, D. S. and Bates, F. C. (1956) A hail size forecasting technique. *Bull. Am. Meteor. Soc.* **37,** 135–41.

Galway, J. G. (1956) The lifted index as a predictor of latent instability. *Bull. Am. Meteor. Soc.* **37,** 528–9.

Hales, J. E. and Crowther, H. G. (1985) Severe thunderstorm cases of 1984. *NOAA Tech. Memo NWS NSSFC-7* (available from NTIS, Springfield, Va., No. PB85-210748/AS).

Hudson, H. R. (1971) On the relationship between horizontal moisture convergence and convective cloud formation. *J. Appl. Meteor.* **10,** 755–62.

Kerr, R. A. (1984) Forecasting of severe storms improved. *Science* **223** (4635), 477.

Leftwich, P. W. Jr. (1985) Verification of severe local storm forecasts issued by the

National Severe Storms Forecast Center, 1984). *NOAA Tech. Memo NWS NSSFC-9* (available from NTIS, Springfield, Va., No. PB86-128105/AS).

Maddox, R. A. and Doswell, C. A., III (1982) Forecasting severe thunderstorms: a brief consideration of some accepted techniques. *National Weather Digest* **7,** 26–30.

Maddox, R. A., Hoxit, L. R., and Chappell, C. F. (1980) A study of tornadic thunderstorm interaction with thermal boundaries. *Mon. Weather Rev.* **108,** 322–36.

Mandics, P. A. and Brown, R. C. (1985) When will the storm get here. *Research & Development* January, 72–7.

McNulty, R. P. (1978) On upper tropospheric kinematics and severe weather occurrence. *Mon. Weather Rev.* **106,** 662–72.

Michie, D., Muggleton, S., Riese, C. E., and Zubrick, S. M. (1984) RuleMaster: a second-generation knowledge engineering facility. *Preprints First Conf. on Artificial Intelligence Applications,* Denver, Colo., IEEE Computer Society and American Association for Artificial Intelligence, pp. 591–7.

Miller, R. C. (1972) Notes on analysis and severe-storm forecasting procedures of the Air Force Global Weather Central. Air Weather Service, *Tech. Report 200* (*Rev.*) (Available from NTIS, Sprintfield, Va., No. Ad-744-042.)

National Oceanic and Atmospheric Administration (1984) *Storm Data* **26,** US Department of Commerce. (Available from National Climatic Data Center, Ashville, N.C.).

Otsby, F. P., Jr (1979) The value of the convective outlook as a planning aid. *Preprints 11th Conf. Severe Local Storms,* Kansas City, Mo., Americal Meteorological Society, pp. 625–7.

Pearson, A. and Weiss, S. J. (1979) Some trends in forecast skill at the National Severe Storms Forecast Center. *Bull. Am. Meteor. Soc.* **60,** 319–26.

Porter, J. M., Means, L. L., Hovde, J. E., and Chappell, W. B. (1955) A synoptic study on the formation of squall lines in the north central United States. *Bull. Am. Meteor. Soc.* **36,** 390–6.

Quinlan, J. R. (1979) Discovering rules by induction from large collections of examples. In *Expert systems in the microelectronic age* (ed. D. Michie). Edinburgh University Press, Edinburgh.

Reynolds, D. W. (1983) Prototype workstation for mesoscale forecasting. *Bull. Am. Meteor. Soc.* **64,** 264–73.

Suomi, V. E., Fox, R., Limaye, S. S., and Smith, W. L. (1983) McIADAS III: A modern interactive data access and analysis system. *J. Climat. Appl. Meteor.* **22,** 766–778.

Uccellini, L. W. and Johnson, D. R. (1979). The coupling of upper and lower tropospheric jet streaks and implications for the development of severe convective storms. *Mon. Weather Rev.* **107,** 682–703.

University Corporation for Atmospheric Research (1982) *National STORM Program: Scientific and Technological Bases and Major Objectives.* Report Prepared for National Oceanic and Atmospheric Administration, under contract NA81RAC00123.

Weiss, S. J., Kelly, D. L., and Schaefer, J. T. (1980) New objective verification techniques at the National Severe Storms Forecast Center. *Preprints 8th Conf. Weather Forecasting and Analysis,* Denver, CO, American Meteorological Society, pp. 412–19.

Weiss, S. J. and Reap, R. M. (1984) Performance characteristics of the TDL automated severe storm outlook: a statistical evaluation. *Preprints 10th Conf. Weather Forecasting and Analysis,* Clearwater, FL, American Meteorological Society, pp. 181–8.

Whitney, L. F., Jr. (1977) Relationship of the subtropical jet stream to severe local storms. *Mon. Weather Rev.* **105,** 398–412.

19

# Comparison of ACLS and Classical Linear Methods in a Biological Application

B. Shepherd
The Turing Institute,
Glasgow, UK
J. Piper and D. Rutovitz
The MRC Clinical and Population Cytogenetics Unit,
Edinburgh, UK

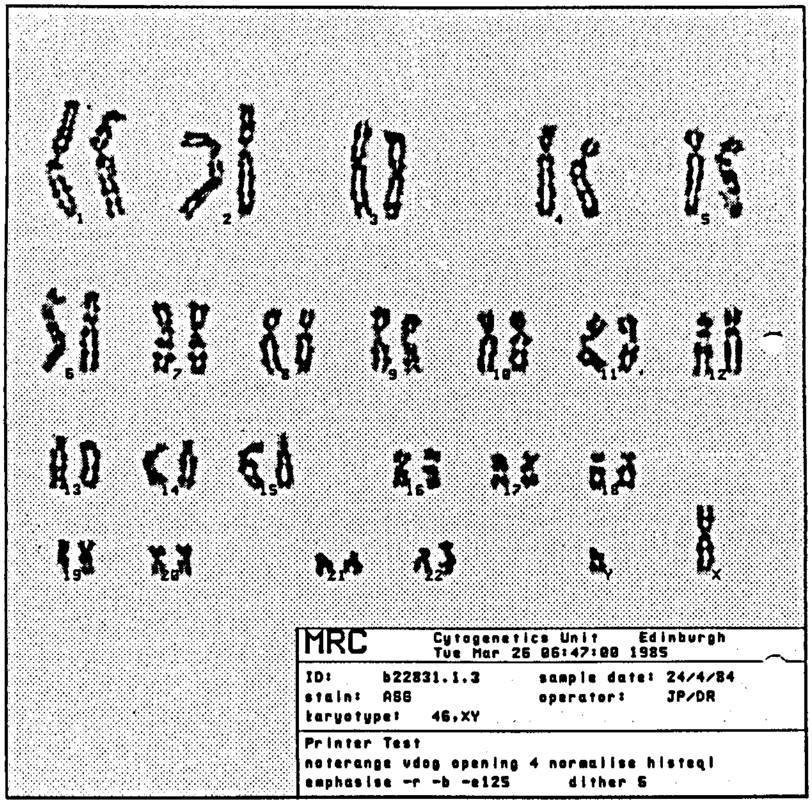
## 1. INTRODUCTION

Classification of hitherto unseen instances of an entity represented by a feature-measurement vector, by extrapolation from findings in the case of a number of classified examples (a 'training-set'), is regarded as a classification problem by statisticians, and as a problem in inductive learning by the Artificial Intelligence community.

Quinlan's ID3 ('Iterative Dichotomiser 3'), a system for inductive learning for discrete-valued features (Quinlan, 1979) was developed and modified by Paterson, Blake and Shapiro and documented by Paterson and Niblett (1983) to deal also with continuously distributed feature measurements; their system is known as ACLS ('Analogue Concept Learning System'). ACLS has been applied to a wide variety of different problems, but has not been tested in many realistic problem domains where the classical approach was well developed and thoroughly evaluated.

Chromosome analysis is such a domain. A considerable amount of work has been put into the problem of classifying the chromosomes in a normal human cell into one of 24 possible classes on the basis of measurements of various features (Piper *et al.*, 1980). Results are available from several different groups working in this domain with linear classifiers. In particular, an extensive data base of chromosome measurements (Figure 1), with the known correct classifications, was available at the Medical Research Council's Cytogenetics Unit in Edinburgh (Piper, 1987).

The performance of the ACLS system and some statistical classifiers were evaluated using this data set subdivided in various ways into learning and test sets. The main conclusions emerging from this study are



```
MRC             Cytogenetics Unit      Edinburgh
                Tue Mar 26 06:47:00 1985
ID:       b22831.1.3        sample date: 24/4/84
stain:    ASG               operator:     JP/DR
karyotype:    46,XY
Printer Test
noterange vdog opening 4 normalise histeql
emphasise -r -b -e125       dither 5
```

Figure 1. A machine-classified metaphase cell, showing representatives of the 24 classes.

the following:

1. A simple linear classifier gave a consistently lower misclassification rate than ACLS.

2. An often claimed advantage for ACLS and similar systems is that the solutions produced can be structured in such a way that makes it easy for the user to understand why the system classifies particular objects as it does (see for example Shapiro and Niblett, 1982). This does not seem to be the case with the types of features and measurements used here, and it is certainly not obvious how to structure the decision-trees to improve intelligibility. If anything, the workings of the statistical classifier are easier to grasp.

3. ACLS scores handsomely in time and memory requirements over classical methods.

4. Because the output of ACLS is a decision-tree which in many cases used only a few feature measurements, it would be possible to use ACLS to guide a programme of analysis so as to calculate only those features

which are required in each particular case. The resultant saving in analysis time might well outweigh the lesser accuracy of the classifier.

5. A modification to the ACLS algorithm designed to prune its generated decision-trees was seen to have beneficial effects on both cost and intelligibility without a major reduction in performance.

## 2. ACLS—AN 'ANALOGUE CONCEPT LEARNING SYSTEM'

2.1. Given a set of measurement vectors and a classification for each of them, ACLS will recursively develop a decision-tree which will correctly classify each of the vectors in the set, except where two or more equal vectors have different class assignments (Figure 2).

2.2. The essential procedure of ACLS can be expressed as follows: consider the feature vectors of which the data set is comprised as representing points in an $N$-dimensional space. ACLS generates a series of decision-planes—hyperplanes perpendicular to individual axes—in an attempt to find compartments which contain elements from only one of the various classes present. The process terminates when all such compartments consist of one class only, or alternatively of identical vectors which belong to more than one class.

2.3. In slightly more detail, the feature vectors of the training set are divided into two classes, namely, those which belong to final compartments (those which have been classified), and those which do not. The latter is termed the residual set. The local residual set is the subset of it which lies in the compartment corresponding to the current position in the decision-tree (the current 'leaf' of the tree). Initially, the decision-tree is empty, and the residual and local residual sets are the training set. At each step the local residual set is examined; if all its members belong to the same class, or are ambiguous, it is not further subdivided. Otherwise it is, by means of a new decision-plane which is defined as follows in the case of the numerical features considered here (termed integer attributes in the ACLS system). Each axis is considered in turn. Elements of the leaf are ordered by the position of their projections on the current axis. The effect of subdivision by a hyperplane perpendicular to this axis, and equidistant between two successive elements of the local residual set is considered. The hyperplane is chosen which gives the 'best' division of the data, where 'best' can be interpreted in various ways. Our notion of 'best' is given below. The corresponding decision

$$x_i > \alpha, \quad x_i \leq \alpha$$

is added to the decision-tree at the current position in it. The two new leaves of the tree which correspond to this division are now dealt with (recursively), and then attention is returned to the preceding node in the tree.

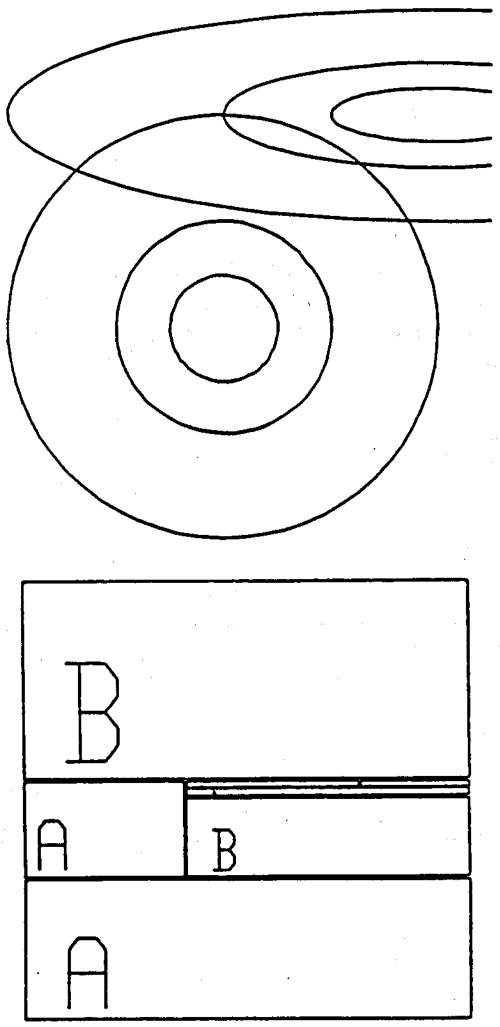


Figure 2. ACLS compartments and Gaussian data: (a) probability contours of two Gaussian distributions; (b) ACLS decision compartments; (c) superimposition of (a) and (b).

2.4. We take 'best' division to mean that which gives the maximum information gain when Shannon's Information Measure is used in the following way. We define the entropy of the residual set $S$ to be

$$H = -\sum p(S) \log p(S)$$

where

$p(S)$ is the prior probability of $S$,

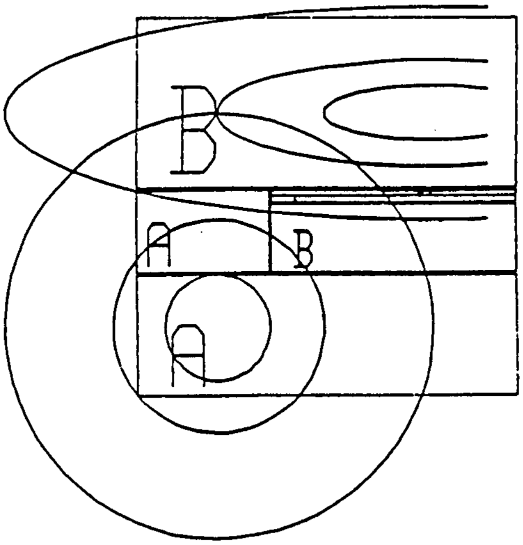


Figure 2 (*continued*)

derived from the probability of an element of $S$ having attribute $a$

$$p(S, a) = \frac{\text{no. of examples which belong to } S \text{ and which have } a}{\text{total number in } S}$$

$$p(S \mid a) = \frac{\text{no. of examples having } a \text{ which belong to } S}{\text{total number having } a}.$$

The entropy gained by splitting on axis $i$ at $\alpha$ is

$$H - H(S \mid A) = H - \sum_S \left\{ \sum_A p(S, a) \log[p(S \mid a)] \right\}$$

where $A$ is the two-attribute set defined by

$$A = \{a \mid (x_i > \alpha), (x_i \leq \alpha)\}.$$

The best split is taken to be one with maximal entropy gain. This is duscussed in more detail in the ACLS user manual (Paterson and Niblett, 1982).

**Termination rules**

If used as described above however, ACLS will continue subdividing the space until every single item in the test set is in a one-class compartment (unless impossible). It is intuitively obvious that beyond a certain granularity of subdivision, additional compartments will subdivide the noise and lengthen the tree without improving performance (see Results section). Many *ad hoc* termination rules have been devised for other algorithms, which generate decision-trees from examples, which are suitable for use with ACLS (see Quinlan, 1983; Seithi and Sarvarayuda, 1982). The one we have used (Shepherd, 1985) is the following: with every learning set we associate a test set which itself is part of the learning process (and ideally should not be the same as that ultimately used to test the effectiveness of the classifier). The ACLS compartmentalization procedure is modified as follows: at every stage of its generation each leaf of the decision-tree is labelled according to the modal class in that leaf. When considering the introduction of a fresh split point, the effect of classifying the elements of the test set according to this rule is investigated. If the new subdivision would result in a worsening of performance rather than an improvement, the split is excluded and the next candidate considered instead.

## 3. DISTANCE AND LIKELIHOOD CLASSIFIERS

Given a family of disjoint sets of a vector space, a 'distance classifier' is one which assigns a vector of unknown class to the set whose mean is closest, distance being measured by a suitable metric. The *Mahalanobis Distance* between an unknown $x$ and a class of mean $y$ is defined by

$$\|x, y\| = (x-y)^{\sim} M^{-1}(x-y)$$

where $M$ is the feature covariance matrix of the class of $y$.

If we can assume that feature values are normally distributed within chromosome classes, the likelihood of $x$ belonging to the class of mean $y$ is

$$L = \frac{1}{(2\pi)^{n/2}\,|M|^{1/2}} \exp\left(-\frac{\|x, y\|}{2}\right)$$

where $n$ is the number of features.

The maximum likelihood classification method has been widely used in chromosome analysis, either on its own or as the first stage of a two-stage classification process, in which the second stage modifies the classification to fit the model which states that there should be just two chromosomes of each class in a normal cell.

Two simplifications are possible

1. If only the trace of the covariance matrix is used, then the resulting classifier is based on variance-corrected distance rather than Mahalanobis distance and is very much cheaper to compute. In Piper (1987) it was shown that the resulting loss of classifier accuracy is rather small, in the region of 1%, and that as a compensation, the classifier may be adequately trained with smaller training sets.

2. If the pooled covariance matrix is used rather than the within-class matrices, the Mahalanobis distance formula can be used as a definition of

distance between any two $x$ and $y$. In this case, at the equiprobability surfaces between two classes $y_i$ and $y_j$

$$(x-y_i)^{\sim}M^{-1}(x-y_i)=(x-y_j)^{\sim}M^{-1}(x-y_j)$$

the $x^{\sim}M^{-1}x$ terms cancel, leaving a linear decision plane, the Fisher Discriminant.

## 4. THE CHROMOSOME CLASSIFICATION DATA SET

This comprises 4200 human chromosomes from 100 cells prepared from peripheral blood samples of 10 different males (prepared according to the ASG technique for G-banding). These had been digitized and, for each chromosome, 28 features measured with the MRC chromosome-measuring system. The calculated class assignments, to one of the 24 classes 1–22, $X$, $Y$ were checked and corrected by an experienced cytogeneticist (Figure 1).

The data was divided into two sets $A$ and $B$, and these were used alternately as training and test sets, and the results averaged.

## 5. RESULTS

The results are given in Tables 1–3.

## 6. DISCUSSION

### Accuracy

The results show that in general, the linear classifier outperforms the ACLS classifier by a generous 10%. Given that ACLS has the capability of correctly classifying everything in the training set, aside from outright clashes (which are very rare) why should there be such a difference? The obvious answer is that ACLS does not treat the noise in the system in an intelligent way: indeed it has to be forcibly restrained (so to speak) from classifying right down into the noise by the tree-pruning process described in para 2.3. If it is fairly obvious that this will result in worthless tree-growing, it is less obvious that it will give worse performance, since ultimately ACLS can draw compartments along any decision surface. Figure 2(a) illustrates what happens when ACLS is given a simple skew-plane problem. The circles and ellipses are schematic probability contours for two normal distributions, A (circles) B (ellipses). ACLS was presented with some random samples from these two distributions. Figure 2(b) shows the first few compartments found and 2(c) shows 2(a) and 2(b) superimposed. The smaller rectangular regions between the two 'B'-labelled regions are a mix of 'A' and 'B', though predominantly 'A' on the left and 'B' on the right. Because ACLS does not work with (multidimensional) probability distributions of data it does not find the intuitively apparent inclined decision-surface separating the classes. It is

Table 1. ACLS results (average of *A* on *B* and *B* on *A*).

| | Unmodified ACLS | | | | Modified ACLS | | | |
|---|---|---|---|---|---|---|---|---|
| No. of features | % correct | Tree size | Av. number of features used | Av. number of features occurring in the tree | % correct | Tree size | Av. number of features used | Av. number of attributes occurring in the tree |
| 11 | 67 | 763 | 9 | 11 | 64.5 | 79 | 5.5 | 10 |
| 17 | 69 | 683 | 9 | — | 66.0 | 80 | 5.5 | — |
| 28 | 72 | 577 | 8 | 25.5 | 69 | 80 | 5.6 | 18.5 |

Table 2. Statistical classifier results.

| | Classification accuracy (%) | | | |
|---|---|---|---|---|
| No. of features | Maximum likelihood *A* on *B* | *B* on *A* | Variance corrected *A* on *B* | *B* on *A* |
| 11 | 80.4 | 80.3 | 77.4 | 79.2 |
| 17 | 83.2 | 83.6 | 80.3 | 83.0 |
| 28 | 82.6 | 81.4 | 83.0 | 85.0 |

Table 3. Summary and comparison: 24-class chromosome class assignment: ACLS *v.* linear comparison.

| | Variance-corrected distance | | (Unmodified) ACLS | | |
|---|---|---|---|---|---|
| No. of features | % correct | Time | % Correct | Av. path length | Time |
| 11 | 78 | 6 s | 67 | 9 | Milli-seconds |
| 17 | 82 | | 69 | 9 | |
| 28 | 84 | | 72 | 8 | |

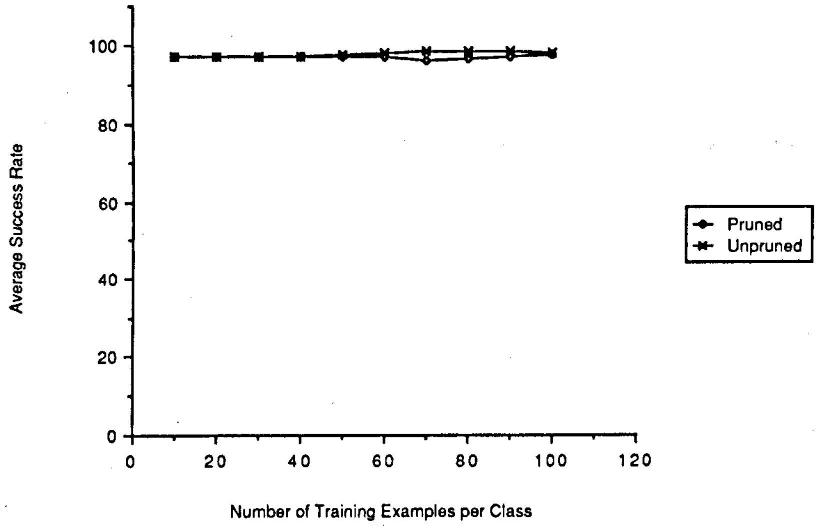


Figure 3. Performance of modified v. unmodified ACLS for a simple 'skew plane problem'.

only concerned with whether its decision-tree is correct so far (with respect to the training set). In consequence it leaves the large B-regions intact, and proceeds to subdivide the small-rectangle area rather finely, thereby achieving a 'perfect' classification of the training set, but with a clearly visible potential for wrongly classifying members of A falling between the 95 and 99 per cent contour lines. The essence of the matter is that ACLS does not know about distributions, and therefore trains itself less efficiently (on regularly distributed data) than a system which does. Thus more data samples may be required to train ACLS to the same level of performance as a linear classifier on linearly separable data (though we do not know if this can always be achieved).

#### Intelligibility

Claims made that ACLS produces classifiers which can be structured so as to be easily intelligible do not seem to apply readily to this case. Consider the following statement:

```
if feature0 < 7845 then
  if feature1 < 7256 then
    if feature2 >= 6379 then
      if feature19 < 8205 then
        if feature10 < 7489 then
          if feature19 >= 7851 then
            if feature4 < 7022 then
              its class18
```

This shows a decision-path leading to classification of chromosomes in one of various classes. Contrast this with the overall linear summing up: '*objects are classified as belonging to a given class if their measurements lie closer to the mean of that class than to the mean of any other*'. This is a subjective matter, but we would be surprised to hear that anybody found a 575-rule decision-tree more enlightening. Caveat: there might be particular problem examples in which an explanation of the decisions was of great interest—for example if there were a significant pattern of non-linear decisions. But it would take an extended study of the decision-trees to reveal such a pattern.

#### Computing resources used

Memory requirements for the ACLS rules and the pooled covariance matrices are of the same order of magnitude; much more storage is required if individual covariance matrices are kept, though by present-day standards nothing of this is significant. The computation time however does matter in the case of the linear systems; by the time all relevant matrix operations are complete, we have used up 0.64 s CPU time to classify a chromosome if the maximum likelihood approach is used, or 0.13 s if the covariance-matrix trace scheme is adopted. ACLS in contrast, has an average rule length of only 9, and the classification time can be under 100 $\mu$s per object.

**Classifier-guided parameter extraction**

The most attractive feature of ACLS from the point of view of this application, is the limited number of parameters involved in any one decision. Very considerable time-savings could be made by using ACLS to guide the parameters called for by the classifier. However, ways of improving ACLS classification accuracy would have to be found, as a 10% difference would waste too much operator time in correcting machine decisions, which would outweigh any savings in machine time realized.

**Pruned v. unpruned ACLS trees**

The particular termination criterion described in para 2.4 has, in this problem domain, led to dramatic reductions in the size of the generated ACLS decision-tree (on average 577 nodes reduced to 80 nodes) with only a small loss in classification accuracy. Although classification accuracy may often be of prime importance, these pruned trees offer substantial reductions in storage costs over their unpruned equivalents. A similar pattern was also seen when the modified ACLS was applied to the 'skew plane problem' described above. Figures 3 and 4 compare the performances and tree sizes of the pruned and unpruned ACLS trees when fed with this 'skew plane data'. It can be seen that the performance of the pruned trees matched that of the unpruned ones, yet their sizes were substantially smaller.

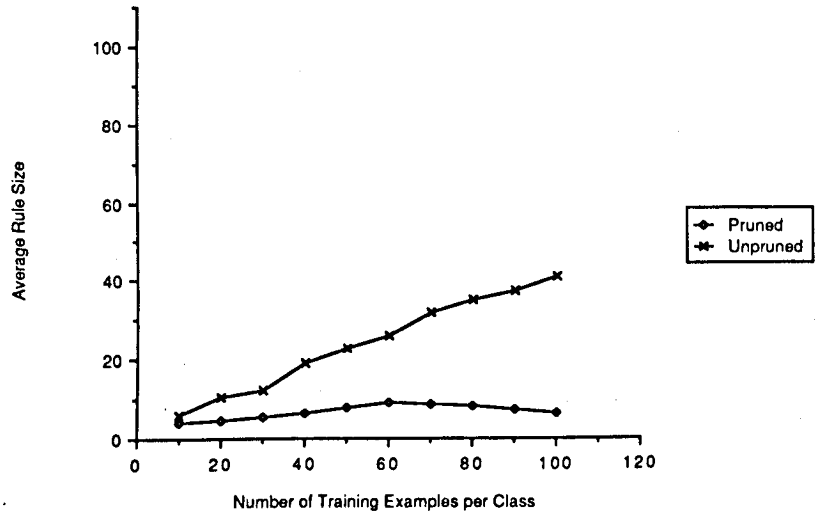


Figure 4. Tree sizes of modified v. unmodified ACLS for a simple 'skew plane problem'.

## REFERENCES

Quinlan, J. R. (1979) Discovering rules by induction from large collections of examples. In *Expert systems in the micro-electronic age* (ed. D. Michie), pp. 168–201. Edinburgh University Press, Edinburgh.

Quinlan, J. R. (1983) Learning from noisy data. *Machine Learning Workshop,* pp. 58–64. Department of Computer Science, University of Illinois, Urbana, Ill.

Paterson, A. and Niblett, T. (1982) *ACLS user manual.* Intelligent Terminals Ltd., Glasgow.

Piper, J. (1987) The effects of zero feature correlation assumption on maximum likelihood classification of chromosomes. *Signal Processing* **12,** 49–57.

Piper, J., Granum, E., Rutovitz, D. and Ruttledge, H. (1980) Automation of chromosome analysis. *Signal Processing* **2,** 203–21.

Seithi, I. K. and Sarvarayuda, G. P. R. (1982) Hierarchical classifier design using mutual information. *IEEE Trans. PAMI,* **4,** 441–5.

Shapiro, A. and Niblett, T. (1982) Automatic induction of classification of rules for a chess endgame. In *Advances in computer chess, 3* (ed. M. R. B. Clarke), pp. 73–91. Pergamon Press, Oxford.

Shepherd, B. A. (1985) Computer induction versus statistical classifiers in the domain of shape recognition. M.Phil thesis, Edinburgh University.

20

# Automatic Synthesis and Compression of Cardiological Knowledge

I. Bratko
E. Kardelj University and J. Stefan Institute
I. Mozetič and N. Lavrač
J. Stefan Institute,
Ljubjana, Yugoslavia

**Abstract**

The paper reports on a study into the mechanical synthesis of the operational knowledge needed for the expert task of electrocardiographic (ECG) interpretation. This knowledge-base was synthesized by means of qualitative simulation based on a causal model of the heart. The resulting (ECG) knowledge-base was subsequently compressed by using inductive learning tools.

## 1. INTRODUCTION

This paper reports on a study into mechanical synthesis of the operational knowledge needed to perform an expert task. The particular task in question is the interpretation of the electrical signals generated by the heart muscle, known as the electrocardiographic (ECG) interpretation. The main contribution of this research is a qualitative model of the electrical activity of the heart which was the basis for mechanical derivation of the ECG diagnostic knowledge.

The heart can be viewed as a mechanical device with an electrical control system. This electrical system works completely autonomously within the heart and is responsible for generating the rhythmical stimulation impulses that cause the contraction of the heart muscle. For proper functioning of the heart, the stimuli have to reach the atria (upper part of the heart) somewhat earlier than the ventricles (lower part of the heart). This is coordinated by the electrical control system which is shown schematically in Figure 1. The contractions of the heart muscle cause changes in the electrical potentials in the body. The changes of these potentials in time can be recorded as an electrocardiograph. Disturbances in the functioning of the heart are reflected in the ECG curves. The interpretation of ECG signals is concerned with the question: if a given ECG curve is not normal, what are the disorders in the heart which could have caused this abnormality?

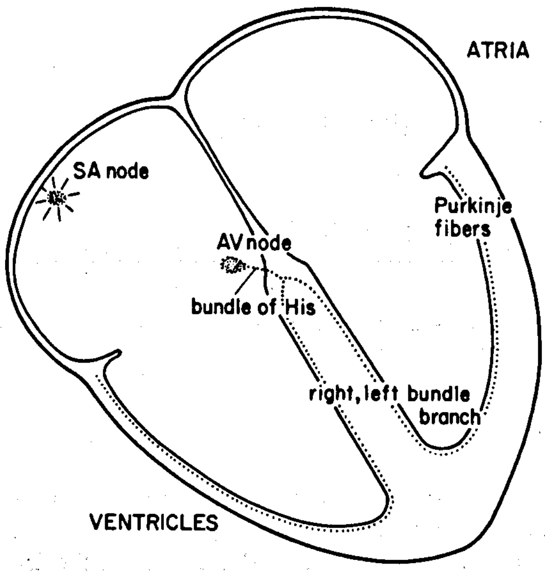


Figure 1. A scheme of the electrical control system of the heart. The nodes generate electrical impulses. The dotted lines represent conduction pathways for impulses.

Various disorders can occur in the electrical control system of the heart. For example, an impulse generator may become silent, or an extra generator may appear, or some electrical conductance may become blocked, etc. These disorders are called *cardiac arrhythmias*. There are about 30 basic disorders and each of them causes some characteristic changes in the ECG. There can be several disorders simultaneously present in the heart. Combined disorders are called *multiple arrhythmias* as opposed to *simple arrhythmias* which correspond to single, basic disorders. The combinatorial nature of arrhythmias complicates the ECG interpretation problem because of the large number of potentially possible combinations. In the medical literature on the cardiac arrhythmias (e.g. Goldman, 1976) there is no systematic description of ECG features which correspond to pairs of simple arrhythmias, let alone triple and even more complicated arrhythmias. On the other hand, these are not very rare in medical practice. In addition to this, multiple arrhythmias are hard to diagnose because there is no simple rule for combining ECGs that correspond to constituent disorders. In other words, if we know which ECGs correspond to any simple disorder, in general it is not clear how to 'sum' these ECGs into 'combined' ECGs corresponding to combinations of simple disorders.

We approached the problem of multiple arrhythmias by constructing a model of the heart. Any combination of disorders can be inserted into the model. The model is *deep* in the context of the distinction between *deep*, causal knowledge, and *shallow*, operational knowledge. By definition, the shallow-level knowledge is sufficient for performing the task itself, but typically without any understanding of the underlying causal mechanisms. The deep knowledge, on the other hand, captures this causal underlying structure and allows the system to reason from first principles.

Our model also is *qualitative* in the sense that it does not deal with electrical signals represented numerically as voltages in time, but represented by symbolic descriptions that specify qualitative features of signals. Such a qualitative modelling approach has several advantages over the conventional numerical modelling. Among the advantages are:

1. The qualitative view is closer to the actual physiological descriptions of and reasoning about the processes and failures in the heart.
2. To execute the model we do not have to know exact numerical values of the parameters in the model.
3. The qualitative simulation is computationally less complex than numerical simulation.
4. The qualitative simulation can be used as a basis for constructing explanations of the mechanism of arrhythmias.

In respect of the qualitative approach to modelling our work is related to the work of Forbus (1984), de Kleer and Brown (1984), and Kuipers (1984).

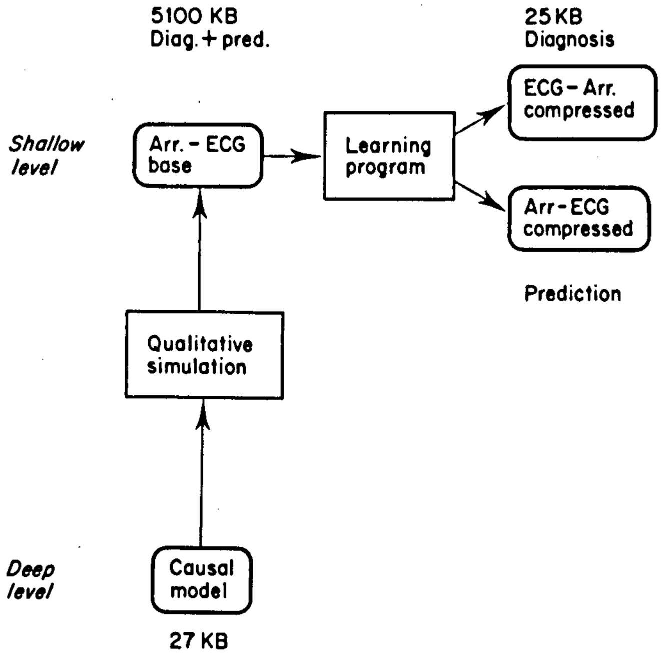


Figure 2. Deep and shallow levels of cardiological knowledge and transformations between these representations.

We used the model for the automatic synthesis (through simulation) of the shallow, operational representation of the ECG interpretation knowledge (see Figure 2). This representation facilitates fast ECG diagnosis, but is rather complex in terms of memory space (about 5 Mbytes). Therefore, as Figure 2 shows, we compressed this knowledge-base by means of inductive learning programs. The representation thus obtained is compact and diagnostically efficient.

In the remainder of the paper we describe the model of the heart, the qualitative simulation algorithm and its efficient implementation, the synthesized shallow knowledge-base and its subsequent compression.

## 2. THE QUALITATIVE MODEL OF THE HEART

Our qualitative model of the electrical activity of the heart specifies causal relationships between objects and events in the heart. These include electrical impulses, ECG signals, impulse generation, impulse conduction and summation. The model can be thought of as an electrical network, as shown in Figure 3. However, signals that propagate in this

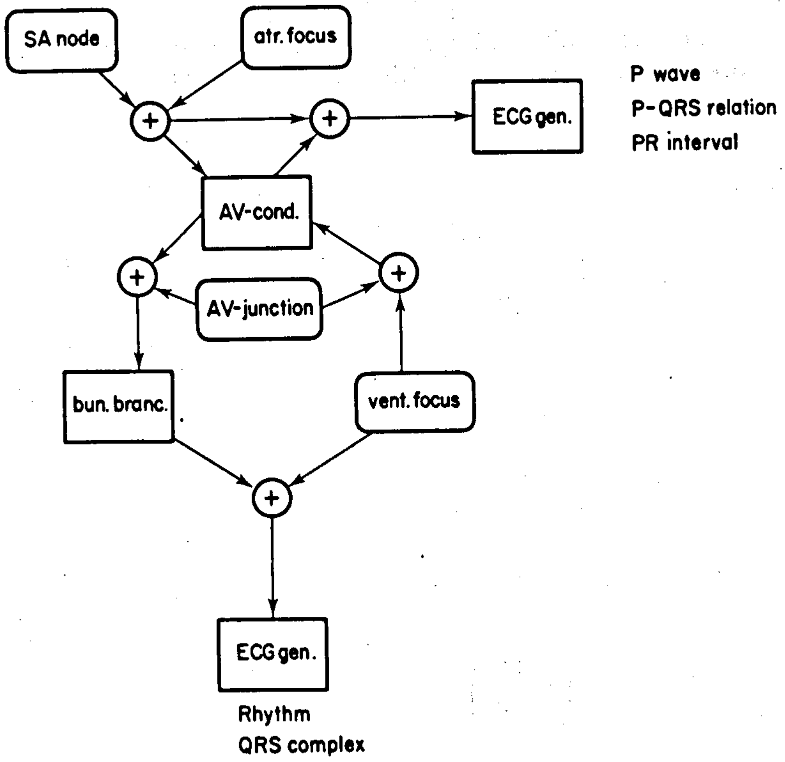


Figure 3. The model of the heart as a network composed of impulse generators, conduction pathways, impulse summators, and ECG generators.

network are represented qualitatively by symbolic descriptions rather than by voltage v. time relations.

The ingredients of the model are: nodes of the network; a dictionary of simple arrhythmias related to heart disorders; 'legality' constraints over the states of the heart; 'local' rule sets; 35 'global' rules.

These ingredients are reviewed in more detail below.

### Nodes of the network

There are four types of nodes: impulse generators, conduction pathways, impulse summators, and ECG generators, illustrated in Figure 4. Recall that the word 'impulse' in this figure refers to a symbolic description, so these elements are in fact operators on descriptions. Impulse generators and conduction pathways can be in normal or abnormal functional states. For example, a generator can generate impulses or can be silent; a conduction pathway can conduct normally or it can be blocked or partially blocked in various ways: it may just cause a delay of an impulse, or it can suppress every second or third impulse, etc. These abnormal states of individual elements correspond to simple disorders of the heart.

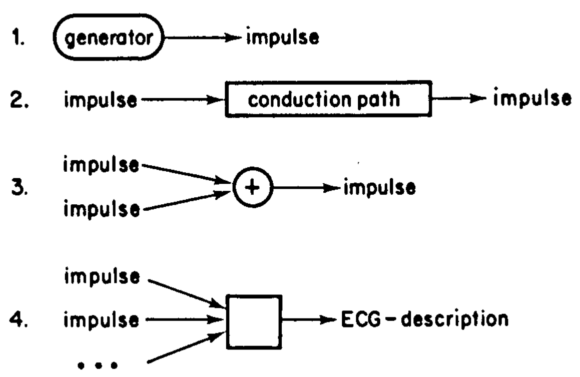


Figure 4. Building blocks for the heart model.

### A dictionary of simple arrhythmias related to heart disorders

Each simple arrhythmia is defined in terms of the functional states of the components of the heart. Roughly speaking, each simple arrhythmia corresponds to a disorder in one of the heart's components.

### 'Legality' constraints over the states of the heart

This is a predicate on the functional states of the heart which recognizes certain categories of states that are rejected by the model as 'illegal'. These categories include: logically impossible states, physiologically impossible states, and 'medically uninteresting' states.

A state is logically impossible if one of the heart's components is in two different states at the same time. An example of a physiologically impossible state is a situation in which two generators in the atria discharge permanent impulses. An example of a 'medically uninteresting' state is one in which there is no atrial activity and the atrio-ventricular(av) conduction is blocked. In such a case the block has no effect on the function of the heart and also cannot be detected in the ECG.

**'Local' rule sets**

These specify the behaviour of the individual components of the heart (generators, summators and conduction pathways) in the presence of various abnormal states.

**'Global' rules**

These rules define causal relations between impulse generators and conduction pathways in the heart, electrical impulses and ECG features; these rules also reflect the structure of the network in Figure 3. There are 35 global rules in the model.

All the rules in the model have the syntax of the first-order predicate calculus, in particular, the syntax that is accepted by PROLOG under Edinburgh notational conventions (Pereira *et al.*, 1978). According to these conventions, the names of constant symbols and functors start with lower-case letters, and the names of variables start with capital letters.

Rules are composed of subexpressions in specialized languages for describing the state of the heart, impulses that are conducted through the heart, and ECG patterns.

For example, the term

```
heart(atr_focus: permanent(regular, between_100_250))
```

is a partial specification of the state of the heart. It says that the atrial focus is discharging permanent impulses (as opposed to periodical) with a regular rhythm at the tachycardic rate (i.e. somewhere between 100 and 250). Each statement about the state of the heart tells in what functional state a component of the heart is (the atrial focus in the example above).

The following is an example of an ECG description:

```
[rhythm = irreglar] &
[regular_P = abnormal] &
[rate_of_P = between_100_250] &
[relation_P_QRS = after_P_some_QRS_miss] &
[regular_PR = prolonged] &
[regular_QRS = normal] &
[rate_of_QRS = between_60_100 or between_100_250]
```

This specification consists of values assigned to qualitative ECG attributes that are normally used in the cardiological literature, such as the rhythm and the shape and the rate of P-waves (Figure 5). Notice that the

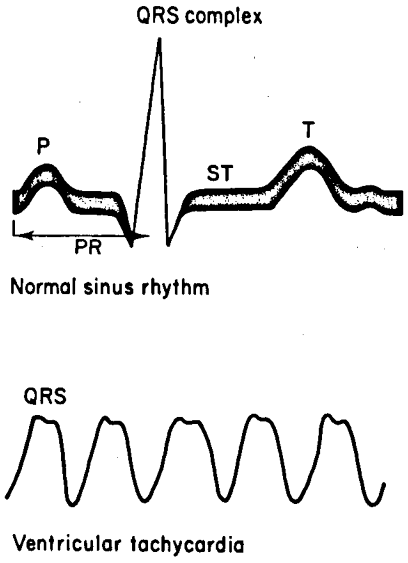


Figure 5. Upper part an ECG curve that corresponds to the normal heart. Marked are features that are normally looked at by an ECG diagnostician. Lower part: ECG curve that corresponds to the arrhythmia ventricular tachycardia. This abnormal ECG is characterized by its higher rate ('tachycardia', between 100 and 250 beats per minute), and the 'wide' shape of the QRS-complexes.

description above gives two values for the rate of QRS waves: it can be either normal (between_60_100) or tachycardia (between_100_250).

Impulses are described by expressions of the form illustrated by the following example:

impulse(atr_focus: form(unifocal, regular, between_100_250))

This says that there are unifocal regular impulses with the tachycardic rate at the atrial focus.

Figure 6 shows two examples of global rules and some rules that specify the behaviour of the individual components of the heart. The first global rule in Figure 6 says:

IF
> the atrial focus discharges permanent impulses at some rhythm Rhythm and rate Rate

THEN
> there will be impulses at the atrial focus characterized by Origin, Rhythm and Rate

WHERE
> Origin, Rhythm and Rate must satisfy the atr_focus relation.

The atr_focus relation describes the behaviour of the atrial focus. This

```
  % Two global rules

[heart(atr_focus: permanent(Rhythm, Rate))]⇒
[impulse(atr_focus: form(Origin, Rhythm, Rate))] &
atr_focus(Origin, Rhythm, Rate).

[impulse(atria: form(_, Rhythm0, Rate0)), heart(av_conduct: State)]⇒
[impulse(av_conduct: form(State, Rhythm1, Rate1))] &
av_conduct(State, Rhythm0, Rhythm1, Rate0, Rate1).

  % Some local relations

atr_focus(unifocal, quiet, zero).
atr_focus(unifocal, regular, between_60_100).
atr_focus(unifocal, regular, between_100_250).
atr_focus(wandering, irregular, between_60_100).
atr_focus(wandering, irregular, between_100_250).
atr_focus(circulating, regular, between_250_350).
  ...

av_conduct(normal, Rhythm, Rhythm, Rate, Rate):-
  below(Rate, over_350).
av_conduct(progress_delayed, regular, irregular, Rate0, Rate1):-
  reduced(Rate0, Rate1).

av_conduct(progress_delayed, irregular, irregular, Rate0, Rate1):-
  reduced(Rate0, Rate1).
  ...

reduced(Rate, Rate).

reduced(Rate, Rate1):-
  succ(Rate1, Rate).

succ(zero, under_60).
succ(under_60, between_60_100).
succ(between_60_100, between_100_250).
  ...
```

Figure 6. Two global rules and some local rules of the heart model. Global rules are, from the point of view of PROLOG, unit clauses of the form: $A \Rightarrow B$ & $C$, which can in the model be read: if $A$ then $B$ where $C$. A special rule-interpreter in PROLOG uses rules of this type. Local rules specify the behaviour of individual components of the heart and are directly executed by the PROLOG system as rules of a PROLOG program.

relation is partially specified in Figure 6 by 'local rules'. It tells that the atrial focus can be quiet, it can behave 'unifocally' discharging impulses at a normal or tachycardic rate with regular rhythm, or it can be 'wandering' discharging impulses with irregular rhythm at normal or tachycardic rate, etc.

The second global rule in Figure 6 can be read:

IF

in the atria there are permanent impulses of some rhythm Rhythm0 and rate Rate0, and the state of the av-conductance is State

THEN

there are impulses of type State, rhythm Rhythm1 and rate Rate1 at the exit from the av-conductance

WHERE

State, Rhythm0, Rhythm1, Rate0 and Rate1 have to satisfy the relation 'av_conduct'.

The av_conduct relation is specified by a set of local rules, as a directly executable PROLOG procedure. This procedure qualitatively defines the physiology of the av_conductance pathway. As can be seen from the definition of this relation in Figure 6, the components of the heart often behave 'non-deterministically' in the sense that they can react to the same input with different responses at the output.

Complete details of the model can be found in Mozetic *et al.* (1984).

## 3. THE QUALITATIVE SIMULATION ALGORITHM

Formally, the qualitative simulation consists of theorem proving and theorem generation. Although the 35 global rules have the syntax of PROLOG they are not directly executed by PROLOG's own interpreting mechanism, the main reason being the necessity for additional control in order to improve the execution efficiency. Thus the qualitative simulation is done by a special rule-interpreter implemented in PROLOG.

Each simulation run consists of the following steps:

1. Instantiate the model by a given arrhythmia, using the definitions of arrhythmias in terms of the heart disorders.
2. Check the resulting functional state of the heart against the legality constraints (logical, physiological, etc.)
3. Execute the model by triggering the rules until no more rules fire (this process is combinatorial due to the non-deterministic nature of the heart's components).
4. Collect the proved assertions about ECG signals and then construct an ECG description that corresponds to the given arrhythmia.

The complex part above is step 3. It is based on the forward chaining of global rules in the model. The simulator starts with some initial data base of facts (initially these just specify the state of the heart) and keeps firing the global rules until no more can fire. The constraint here is that no rule is repeatedly executed on the same piece of information. Execution of rules generates new assertions that are added into the data base. These new assertions are regarded as hypotheses that can later be proved false.

1. By the definition of [atrial_tachycardia, wenckebach] instantiate the state of the heart to:

   heart(sa_node: permanent(quiet, zero)) &
   heart(atrial_focus: permanent(regular, between_100_250)) &
   heart(av_conduct: progress_delayed) &
   ...

2. The assertion

   heart(atrial_focus: permanent(regular, between_100_250))

   triggers the first global rule in Figure 6. The goal

   atr_focus(Origin, regular, between_100_250)

   is evaluated, using the local relation atr_focus. This succeeds and the new assertion is added to the database:

   impulse(atr_focus: form(unifocal, regular, between_100_250))

3. After summing together the atrial impulses we get the asertion:

   impulse(atria: form(unifocal, regular, between_100_250))

4. This assertion triggers the second global rule in Figure 6. The Prolog goal to be evaluated is now:

   av_conduct(progress_delayed, regular, Rhythm1, between_100_250, Rate1)

   This can be satisfied in two ways, either by Rate1 = between_100_250 or by Rate1 = between_60_100. So two hypotheses are indicated:

   impulse(av_conduct: form(progress_delayed, irregular, between_100_250))

   or

   impulse(av_conduct: form(progress_delayed, irregular, between_60_100))

   Depending on the search strategy used, the system may now assert one of the above hypotheses into the database, and consider the second one on backtracking which corresponds to a depth-first style search; or, it may assert both which corresponds to a breadth-first style search.

Figure 7. Fragments of the qualitative simulation trace for the combination of arrhythmias atrial_tachycardia and wenckebach. Some steps were omitted from the actual trace.

Backtracking to a previous point occurs when the current content of the data base is found inconsistent, i.e. some assertion has been generated which leads to contradiction. Roughly, the rule triggering process is as follows.

Assume that there is a hypothesis $A$ in the data base. Then apply a global rule of the form

$$A \Rightarrow B \ \& \ C$$

In general, in such rules $A$ and $B$ are PROLOG terms and $C$ is a PROLOG goal which can directly be executed by PROLOG. The precondition matching is simply the logic unification. Normally, $C$ is a call to evaluate a local relation. Thus, to apply a rule of the above form, do:

*Evaluate C; if C is false then A must be false and discard it; otherwise if C is true, assert a new hypothesis B and continue firing rules.*

In the case that $C$ is false a contradiction has been detected and backtracking is indicated. This process terminates when there are no more rules to fire. At that stage, all the remaining hypotheses in the data base are accepted as true since there is now no way of showing a contradiction. Among these facts there are also statements about the ECG. The simulator collects those statements and forms an ECG description which corresponds to the arrhythmia with which the simulation process was started.

As an example, Figure 7 shows part of a simulation run when the state of the heart is a combination of two simple arrhythmias: *atrial_tachycardia* and *wenckebach.*

## 4. IMPLEMENTATION OF THE SIMULATION ALGORITHM

The easiest way of implementing the simulation algorithm outlined above is to use the depth-first search strategy. This is straightforward and suitable for single simulation runs, that is for answering questions of the *prediction* type: given an arrhythmia, what are its corresponding ECGS. Different possible ECGS are simply generated through backtracking. Also, an execution trace obtained in such a simulation run can be used as the basis for generating a user-oriented explanation of what is going on in the heart. This is suitable since the simulation steps follow the causal chains of events in the heart, according to the global rules of the model. These rules essentially describe the causal relations between events in the heart.

In a PROLOG implementation of depth-first simulation on the DEC-10 (Edinburgh implementation of PROLOG; Pereira 1978), each simulation run takes a few c.p.u. seconds, producing all alternative ECGS.

*Diagnostic-type* questions as opposed to *prediction-type* questions, are of the form: given an ECG, what arrhythmias could have caused it? To answer such questions, we could run the model in the opposite direction. Start with a given ECG and end with the possible functional states of the heart that might cause this ECG.

We can in fact run the model in this direction by the backward chaining of the rules in the model. In order to do that we reversed the global rules and used the simple depth-first search. However, the practical problem of efficiency now arises because the branching factor ('non-determinism') in the backward direction is much higher than that in the forward direction. This entails much more backtracking and rather complex search, thus rendering this approach to diagnosis impractical. Efficiency can be improved by re-writing the model so as to introduce more constraints into the rules, which helps the system recognize contradictory branches at an earlier stage. An attempt at re-formulating rules, however, revealed two drawbacks. The size of the model increases considerably, and the transparency is greatly affected. This, in turn, mars the explanation of the heart's behaviour based on the execution trace.

An alternative way to achieve efficient diagnosis is to generate from the deep model of the heart a complete shallow-level representation of the arrhythmia–ECG relation as a set of pairs of the form:

(Arrhythmia, ECG-description)

In principle this can be done by executing the depth-first simulation (forward chaining) for each possible combined arrhythmia, and storing all its ECG manifestations. It would be necessary to repeat this for all possible alternative execution paths in order to obtain all possible ECGs for each arrhythmia. This is again rather inefficient for two reasons. First, for each disjunctive solution the simulator has to backtrack to some previously used rule in the model and restore its previous state. Second, the final resulting ECG descriptions have the form of disjunctions of ECG patterns. These disjunctive expressions can be more complex than necessary and can be later simplified. This posterior simplification, however, is again a complex operation. Each disjunct is the result of an alternative execution path. The simplification can be carried out much more economically at the very moment that a disjunct (or, typically part of it) is generated, before it is further expanded and mixed in the expression with other not closely related terms.

These two factors (saving the restoration of previous states, and immediate simplification of disjunctive expressions) motivated the implementation of another type of simulation algorithm which handles alternative execution paths in a breadth-first fashion. This algorithm develops alternatives essentially in parallel and currently simplifies disjunctions. The simplification rules actually used are rather model dependent in the sense that they do not preserve logical equivalence in general but only in the special case of the properties of the heart model.

So the 'breadth-first' simulation is not general and we would possibly have to modify the simplification rules in the case of a change in the model.

This specialized simplification method proved to be rather powerful. As a typical example of the reduction effect, consider the combined arrhythmia *atrial_fibrillation and ventricular_ectopic_beats.* The depth-first simulation generates 72 ECG descriptions which corresponds to an ECG expression with 72 disjunctive terms. The breadth-first simulation results in a description comprising four disjunctive terms. There was a similar factor of improvement in general, which can be seen from the results of generating the complete arrhythmia–ECG relation.

## 5. GENERATION OF A COMPLETE ARRHYTHMIA–ECG KNOWLEDGE-BASE

The breadth-first simulation algorithm was executed on all mathematically possible combinations of simple arrhythmias. The majority of these combined arrhythmias were eliminated by the legality constraints over the states of the heart. The complete arrhythmia-ECG knowledge-base was thus automatically generated. Results of the generation are depicted in Figure 8.

Figure 8 reveals some interesting points. Of all possible arrhythmias, the combinations of four simple arrhythmias are the largest subset. Note the large number (140,966) of generated ECG patterns. This is indicative of the difficulty in ECG diagnosis of cardiac arrhythmias. On average, each arrhythmia has almost 60 different corresponding ECG manifestations. There are altogether 2419 'legal' combined arrhythmia within the level of

| Number of disorders in the heart | Mathematically possible combinations | Numbers of generated | | |
|---|---|---|---|---|
| | | Multiple arrhythmias | Prolog clauses | ECG descriptions |
| 1 | 30 | 18 | 27 | 63 |
| 2 | 435 | 118 | 286 | 2,872 |
| 3 | 4,060 | 407 | 1,207 | 17,551 |
| 4 | 27,405 | 759 | 2,679 | 45,939 |
| 5 | 142,506 | 717 | 2,867 | 52,707 |
| 6 | 593,775 | 340 | 1,164 | 20,322 |
| 7 | 2,035,800 | 60 | 84 | 1,512 |
| Σ | 2,804,011 | 2,419 | 8,314 | 140,966 |

Figure 8. Results of generating the arrhythmia knowledge-base. The number of generated arrhythmias for 'combinations' of simple arrhythmias is 18 which is less than the number of all simple arrhythmias (30). The reason is that some simple arrhythmias (conduction disturbances and ectopic beats) cannot occur alone, but only in combination with other arrhythmias (e.g. with sinus rhythm).

detail of the heart model. The relation to their ECG manifestations is represented by 8314 PROLOG clauses. Each clause represents a pair: arrhythmia–ECG expression. Each ECG expression specifies a number of possible ECGS, about 20 on average. This is the reduction factor due to the simplification technique used in the breadth-first simulation.

The set of 140,966 ECG patterns (the right-hand sides of the arrhythmia–ECG rules) are not unique. The same ECG patterns can occur at several places which means that several arrhythmias can have the same ECG manifestation. Consequently, arrhythmias cannot be unambigously diagnosed from a given ECG. Empirical probing showed that for a given ECG there are typically between two and four possible combined arrhythmias in the arrhythmia knowledge-base. From the medical point of view, however, these alternative diagnoses are not significantly different in the view of treatment. They would typically all require the same treatment.

The arrhythmia–ECG base generated from the model is complete in two ways. First, it comprises all physiologically possible arrhythmias at the level of detail of the model. Second, each arrhythmia is associated with all its possible ECG manifestations. In principle, the problem of diagnosing is now simple. As the rules in the knowledge-base are logical implications, we can apply *modus tollens* rule of inference on them. Consider a rule of the form

Arrhythmia ⇒ ECG_description

where ECG_description is the disjunction of *all* possible ECGS that Arrhythmia can cause. Then, if a given ECG does not match ECG_description it follows that Arrhythmia is eliminated as a diagnostic possibility. All arrhythmias that are not thus eliminated form the set of possible diagnoses with respect to the given ECG data. Any further discrimination between the set of arrhythmias thus obtained can be done only on the basis of some additional evidence (e.g. clinical data). Also, as the knowledge-base is complete the empty set of possible arrhythmias would imply that the given ECG is physiologically impossible.

## 6. COMPRESSION OF THE ARRHYTHMIA–ECG BASE USING INDUCTIVE LEARNING TOOLS

The main motivation for having the arrhythmia–ECG base is that it can be used for ECG diagnosis based on a simple pattern-matching rule. However, it is rather bulky for some practical application requirement. If stored as a text file, the 8314 PROLOG clauses that represent the arrhythmia–ECG relation, occupy 5.1 Mbytes of store. Also its complexity renders this knowledge-base difficult to compare with the conventional medical codifications of electrocardiographic knowledge. Therefore an attempt has been made to find a more compact representation of the arrhythmia–ECG base that would still allow efficient ECG diagnosis.

The main idea was to use the knowledge-base as a source of examples of particular features (heart disorders or ECG features) and to use an inductive learning algorithm to obtain their compact descriptions. The inductive learning programs used were GEM (Reinke, 1984) and EXCEL (Becker, 1985). Taking the complete arrhythmia–ECG base as the set of examples, the number of examples for these two learning programs would be too high. Therefore we had first to generate a subset of the knowledge-base that would retain its completeness to the greatest possible extent. The following domain-specific factorization properties facilitated the selection of a considerably reduced subset for learning, whereby the information thus lost can be recovered by a small set of additional rules.

Some disorders in the heart are of a permanent nature while some do not occur regularly; the latter are called ectopic beats. A large number of combined arrhythmias and in particular ECG descriptions are due to the unconstrained combinatorial nature of ectopic beats. If we disregard mutual combinations of different types of ectoptic beats we can substantially reduce the number of generated multiple arrhythmias and ECG descriptions. The information thus lost can easily be reconstructed from the remaining rules in the knowledge-base. Namely, different types of ectopic beats are both mutually independent and independent of permanent disorders. The presence or absence of an ectopic beat does not affect the part of the ECG description produced by other disorders. The learning subset of the knowledge-base was further reduced by disregarding three simple arrhythmias whose ECGs can easily be deduced from the behaviour of other similar arrhythmias.

To summarize—the learning subset was constructed from the original arrhythmia-ECG base by the following reductions:

1. The subset only deals with 27 simple arrhythmias instead of the original repertoire of 30. We omitted *sinus_arrhythmia, right_bundle_branch_block,* and *multi_ventricular_ectopic_beats* whose ECG description can be constructed from the descriptions of sinus_node_disorders, left_bundle_branch_block, and ventricular_ectopic_beats respectively.

2. We discarded mutual combinations of different types of ectopic beats (atrial_ectopic_beats, junctional_ectopic–beats, and ventricular_ectopic_beats).

The subset thus obtained was substantially smaller than the original arrhythmia–ECG base. There are 586 combined arrhythmias and 2405 ECGs in the subset compared with 2419 arrhythmias and 140,966 ECGs in the complete knowledge-base.

Roughly, the procedure for compressing the knowledge now proceeded as follows. The learning subset of the arrhythmia–ECG base comprised

586 rules (corresponding to 586 combined arrhythmias) of the form:

Combined_arrhythmia ⇒ ECG_description.

The goal of learning was to convert this information into rules of two forms:

1. Compressed *prediction rules* which answer the question: what ECGS may be caused by a given disorder in a heart's component?
2. Compressed *diagnostic rules* which answer the question: what heart disorders are indicated by a given isolated ECG feature?

Compressed prediction rules were synthesized by the GEM inductive learning program, and compressed diagnostic rules were synthesized by the EXCEL program. Both programs are based on the AQ11 learning algorithm (Michalski, 1983) and both generate class descriptions as APC expressions (Annotated Predicate Calculus; Michalski, 1983). Before the programs could be used, the learning subset had to be converted into rules of yet two other forms in order to obtain the right input for the learning programs required, i.e. examples of objects that belong to classes being learned. The principle of how to do that in general is described in Mozetic (1986). For synthesizing prediction rules, the proper starting form was:

Isolated_disorder ⇒ ECG_description

where an 'isolated disorder' is, for example, the atrial focus being in the tachycardic state. The starting point for the synthesis of diagnostic rules were rules of the form:

Isolated_ECG_feature ⇒ Heart_state_description

where an 'isolated ECG feature' is, for example, P-wave having abnormal shape. In general, rules of these forms are not completely logically equivalent to the original rules, so they have to be used with care. Mozetic (1986) states the conditions under which both the original rules

| Total number of | Original arrhythmia knowledge-base | Subset of the knowledge-base | Arrhythmias combined | Diagnostic rules |
|---|---|---|---|---|
| rules | 2,419 | 586 | 45 | 49 |
| conjunctions | 8,314 | 957 | 75 | 144 |
| attributes | 58,197 | 6,699 | 248 | 371 |
| Kbytes | 5,100 | 400 | 10 | 13 |

Figure 9. Comparison between the original arrhythmia–ECG base, the selected subset for learning, and the derived compressed rules of both types (prediction and diagnosis). A 'rule' above corresponds to a combined arrhythmia, a 'conjunction' corresponds to a PROLOG clause. 'Attributes' mean all the references to attributes in a whole rule set. The last row gives the sizes of these representations if stored as text files.

and the transformed ones are in fact logically equivalent. So this additional request had to be verified in our case as well.

Figure 9 shows the compression effects achieved in terms of the number and complexity of rules, and in terms of storage space needed when storing different representations simply as text files.

Figure 10 shows some prediction and some diagnostic rules generated

```
combined(wenckebach) ⇔ [av_conduct = wen] ⇒
   [relation_P_QRS = after_P_some_QRS_miss] &
   [regular_PR = prolonged]

combined(atrial_tachycardia) ⇔ [atr_focus = at] ⇒
   [regular_P = abnormal] &
   [rate_of_P = between_100_250] &
   [regular_PR = meaningless ∨ normal ∨ prolonged]
   ∨
   [regular_P = abnormal] &
   [regular_PR = shortened ∨ normal ∨ prolonged] &
   [regular_QRS = wide_LBBB_RBBB ∨ delta_LBBB ∨ delta_RBBB]
   ∨
   [regular_P = abnormal] &
   [rate_of_P = between_100_250] &
   [regular_PR = shortened] &
   [regular_QRS = normal ∨ wide_LBBB ∨ wide_RBBB]

[regular_P = abnormal] ⇒
   [sa_node = quiet] &
   [atr_focus = quiet ∨ at ∨ afl ∨ af ∨ aeb] &
   [reg_vent_focus = quiet ∨ vr ∨ avr ∨ vt]

[regular_PR = shortened] ⇒
   [atr_focus = quiet ∨ wp ∨ at ∨ mat ∨ aeb] &
   [av_conduct = wpw ∨ lgl]
   ∨
   [av_conduct = normal] &
   [av_junction = jb ∨ jr ∨ jt]
   ∨
   [atr_focus = at] &
   [av_conduct = normal ∨ wpw ∨ lgl]

[regular_QRS = normal] ⇒
   [av_conduct = normal ∨ avb1 ∨ wen ∨ mob2 ∨ avb3 ∨ lgl] &
   [bundle_branches = normal] &
   [reg_vent_focus = quiet]
```

Figure 10. Examples of prediction and diagnostic rules generated by the inductive learning programs.

by the induction algorithms. Some of these descriptions correspond very well to the definitions in the medical literature. For example, the descriptions of the *wenckebach* disorder corresponds precisely to the conventional medical descriptions. On the other hand, some of the synthesized descriptions were considerably more complex than those in the medical literature. The computer-generated descriptions in such cases give much more detailed specification than may be necessary for an intelligent reader with a physiological background. Such a reader can usually infer the missing detail from his background knowledge. The additional details must still be made explicit in case of a computer application in the form of a diagnostic expert system, otherwise a lot of background knowledge and inference would have to be added which would be extremely difficult and its correctness hard to verify. Mozetič (1986) describes in detail how the knowledge compression was done.

## 7. CONCLUSIONS

Various representations of the ECG knowledge and transformations between these representations were described. The main three representations are at different knowledge-levels in the sense of a distinction between 'deep knowledge' (causal, first principles) and 'shallow knowledge' (operational knowledge). These three representations are: deep level (the qualitative causal model of the heart); shallow level (the complete arrhythmia–ECG base); and shallow level compressed (compact diagnostic and prediction rules).

Figure 11 compares these representations from the points of view of: nature of knowledge, method of construction, representational formalism, size as text file (in kilobytes), direction of inference the representation supports, functional role.

The size of the compressed diagnostic knowledge of 25 kbytes apparently contradicts the size of compressed diagnostic rules in Figure 9. The difference stems from the fact that the compressed *rules* themselves are not sufficient for the diagnosis because of the loss of information in the selection of the learning subset of the arrhythmia–ECG base. To attain the full diagnostic equivalence with the complete shallow knowledge-base, we have to add descriptions of arrhythmias and ECG features that were eliminated when reducing the complete knowledge-base into the learning subset. Furthermore, 'legality' constraints and related testing procedures must also be included. After adding all of these, the size of the compressed diagnostic knowledge increases to 25 kbytes.

The ECG knowledge of this study is used in various forms in the KARDIO expert system for ECG interpretation (Lavrac *et al.*, 1985). The automatically synthesized shallow-level electrocardiographical knowledge-base is complete with respect to the level of detail of the model. In an

| | I<br>causal model<br>of the heart | II<br>arrhythmia<br>knowledge-base | III<br>compressed<br>diagnostic knowledge |
|---|---|---|---|
| Nature of knowledge | deep causal | shallow operational | shallow operational |
| Method of construction | manual | automatic synthesis from I | automatic compression from II |
| Representational formalism | first-order logic | propositional logic | propositional logic |
| Size in kbytes | 27 | 5,100 | 25 |
| Direction of inference | ARR→ECG forward | ARR↔ECG both | ARR←ECG backward |
| Role | qualitative simulation generate II | diagnosis, provide examples for III | diagnosis |

Figure 11. Comparison of different representations of electrocardiological knowledge.

assessment study of KARDIO (Grad and Cercek, 1984), cardiologists made the following estimates: the knowledge-base covers 90–95% of a 'non-selected' patient population suffering from cardiac arrhythmias (*non-selected* in the sense that these patients would not be referred to a specialist cardiologist on the account of previous examinations). In a *selected population* KARDIO-E (the version of KARDIO used in this assessment study) would correctly handle 75% of arrhythmia cases. In an actual test on 36 randomly selected arrhythmia cases from internal medical practice the arrhythmia knowledge-base was sufficient in 34 cases (94%). The failed cases are due to some incompleteness of the deep model, such as the present model's incapability to handle artificial pacemakers.

The main vehicles for implementing various knowledge representations and transformations were the following tools and techniques of Artificial Intelligence: logic programming (PROLOG in particular), qualitative modelling, and inductive learning tools. It should be noted, of course, that the inductive programs were used in this work as tools for compression of a representation and not for actual learning. Since the input information to the learning programs was complete no generalization could have occurred.

Further work can be directed along various lines including: elaboration of explanation capabilities based on the qualitative simulation; extending the model of the heart with treatment of mechanical failures; and stratifying the model by introducing several levels of abstraction. This could be based on hierarchical relations between components of the heart and attribute values. Such a hierarchy would have an important role in generating a good and concise explanation of the heart to provide a means of flexibly concentrating the explanation on points selected by the user.

**Acknowledgements**

The authors would like to thank the cardiologists: Professor M. Horvat, B. Cercek, A. Grad, and P. Rode of the University Medical Centre, Ljubljana, for their consultations as domain specialists in this research.

## REFERENCES

Becker, J. (1985) Inductive learning of decision rules with exceptions. M.Sc. thesis. University of Illinois at Urbana-Champaign.

Forbus, K. D. (1984) Qualitative process theory. *Artificial Intelligence* **24**, 85–168.

Grad, A. and Cercek, B. (1984) *Evaluation of the applicability of the* KARDIO-E *expert system.* Issek Workshop 84, Bled, Yugoslavia.

Goldman, M. J. (1976) *Principles of clinical electrocardiography.* Lange Medical Publications, Los Altos.

Kleer, J. de and Brown, J. S. (1984) A qualitative physics based on confluences. *Artificial Intelligence* **24**, 7–84.

Kuipers, B. (1984) Commonsense reasoning about causality: deriving behaviour from structure. *Artificial Intelligence* **24**, 169–204.

Lavrac, N., Bratko, I. Mozetic, I., Cercek, B, Horvat, M. and Grad, A. (1985) KARDIO-E—an expert system for electrocardiographic diagnosis of cardiac arrhythmias. *Expert Systems* **2**, 46–50.

Michalski, R. S. (1983) A theory and methodology of inductive learning. In *Machine learning—an artificial intelligence approach* (eds R. S. Michalski, J. G. Carbonell and T. M.Mitchell) pp. 83–134. Tioga, Palo Alto.

Mozetic, I. (1986) Compression of the ECG knowledge-base using the AQ inductive learning algorithm. Report no. UIUCDCS-F-85-943, Department of Computer Science, University of Illinois at Urbana-Champaign.

Mozetic, I. (1986) *Knowledge extraction through learning from examples.* In *Machine learning; a guide to current research* (eds T. M. Mitchell, J. G. Carbonell, and R. S. Michalski) pp. 227–31. Kluwer, Boston.

Mozetic, I., Bratko, I. and Lavrac, N. (1984) *The derivation of medical knowledge from a qualitative model of the heart.* ISSEK Workshop 84, Bled, Yugoslavia. Updated version to appear as *KARDIO: a study in deep and qualitative Knowledge for expert systems,* MIT Press, Cambridge, Mass.

Pereira, F., Pereira, L. M., and Warren, D. H. D. (1978) *DecSystem-10 Prolog user guide.* Department of Artificial Intelligence, University of Edinburgh, Edinburgh.

Reinke, R. E. (1984) Knowledge-acquisition and refinement tools for the ADVISE META-EXPERT system. M.Sc. thesis, University of Illinois at Urbana-Champaign. Also appeared as UIUCDCS-F-84-921.

# Index