17

Knowledge-based programming self-applied

C. Green* and S. J. Westfold*†
Systems Control Inc.
Palo Alto, California

Abstract

A knowledge-based programming system can utilize a very-high-level self description to rewrite and improve itself. This paper presents a specification, in the very-high-level language V, of the rule compiler component of the CHI knowledge-based programming system. From this specification of part of itself, CHI produces an efficient program satisfying the specification. This represents a modest application of a machine intelligence system to a real programming problem, namely improving one of the programming environment's tools – the rule compiler. The high-level description and the use of a programming knowledge base provide potential for system performance to improve with added knowledge.

1. INTRODUCTION

This paper presents a specification of a program in a very-high-level description-oriented language. Such a specification is suitable for compilation into an efficient program by a knowledge-based programming system. The program specified is the rule compiler component of the CHI knowledge-based programming system (Green *et al.* 1981). The language used for the specification is V, which is used in CHI to specify programs as well as to represent programming knowledge (see Phillips 1982). The compiler portion of CHI can produce an efficient program from this self description. The availability of a suitable self description allows not only self compilation, but also enhanced potential for the knowledge-based system to assist in its own modification and extension.

We use the term 'knowledge-based programming system' to imply that most of the programming knowledge used by the system is expressed explicitly, usually in some rule form, and is manipulable in that form. This collection of programming rules is used by the system to help in selecting implementation techniques, and

* Present address: Kestrel Institute, Palo Alto.
† Also with Computer Science Department, Stanford University.

to help in other programming activities such as editing and debugging. By comparison a conventional compiler tends to use procedures that compile source language constructs into preselected choices for data and control structure implementations rather than exploring alternative implementations.

The programming knowledge base includes stepwise refinement or transformation rules for optimizing, simplifying and refining data structures and control structures. The synthesis paradigm of CHI is to select and apply these rules to a program specification, generating a space of legal alternative implementations. By applying different refinement rules from CHI's knowledge base in different orders one gets alternative implementations, whose efficiency characteristics can be matched to the problem. In general, refinement choices may be made interactively by the user or automatically by incorporating efficiency estimation knowledge (see Kant 1979 for a discussion of this important issue). It is our intent that strategies for rule selection and ordering will be expressed in the meta-rule language portion of V. But in this paper we just present a particular set of rules and an order of application rather than alternative synthesis rules and strategies for selecting among them.

The V language is used not only to specify programs but also to express the knowledge base of synthesis rules and meta-rules. The high-level primitives of V include sets, mappings, relations, predicates, enumerations and state transformation sequences. It is a wide-spectrum language that also includes low-level constructs. Both declarative and procedural statements are allowed. In this language there is little distinction between the terms 'program specification,' 'program description' and just 'program'. We use the terms interchangeably. V is translated into LISP by the compiler portion of CHI.

In choosing a program for use as a start on self modification, we decided to work with a program that is being used and modified, rather than a contrived example. We picked RC, a *rule compiler* written in LISP that compiles the production rule subset of the V language into efficient LISP code. Since the program refinement rules are expressed as production rules, the rule compiler allows refinement rules to be expressed in a simple, readable, and concise syntax without loss of efficiency. By expressing refinement rules in a clean formalism, their content is more readily available for scrutiny, improvement, and transfer to other systems. Since production rules are also a method of specifying programs, it is possible to specify the rule compiler using this production rule subset of V. Thus the very-high-level version of the compiler can be tested by compiling itself.

We have succeeded in creating a very-high-level description of RC in the V language. The remaining sections of this paper present this description. The adequacy of the description has been tested by having RC compile itself. More precisely, the original version of RC in LISP compiled the V description of RC into LISP. This newly-compiled LISP program was then tested by having it compile its V description.

We decided that rather than attempting to create an ideal version of the rule

compiler immediately, we would first create and present a V version that approximates the input-output performance of the original program. The original program was written in LISP in 1979 as part of the first version of CHI and had some undesirable limitations in its input formats and optimizations. The V version is being extended to overcome the limitations.

The very-high-level V version proves to have several advantages over the LISP version: size, comprehensibility, and extensibility. The V version is less than 20% of the size of the LISP version (approximately 2 pages versus 10 pages). The size improvement is due in part to a better understanding of the program, but is mainly due to the declaratively-oriented description and high-level constructs that are allowed in the very-high-level language, and the concomitant allowance of other general progamming knowledge to fill in details that needed to be explicitly provided in the LISP version. Rules compiled by the V version of RC are more efficient than those produced by the LISP version of RC, due to some simplification rules in CHI.

Subjectively, we find RC in V much easier to understand and extend. In section 3 of this paper we present several possible extensions to the rule compiler, and discuss how our approach facilitates these extensions. Indeed, the difficulty of making frequently-required extensions was a reason for using the rule compiler in this study. A major step would be the extension of the description of the small compiler RC to a description of the entire V compiler portion of CHI. Then the rule compiler will not be a separate component, but will be merged with the knowledge-based compiler portion of CHI. Rules will then be compiled as is any other V program.

Is the self-application of CHI really different from that done in other systems? It appears to differ to an extent that may make a difference. Obviously self-referencing is possible in many languages, from machine language up, and bootstrapping is often done with compilers. The notion of a language with an accessible, sophisticated environment expressed in the same language already occurs in SMALLTALK (Ingalls 1978), INTERLISP (Teitelman & Masinter 1981) and other languages. These systems provided much of the inspiration for our work. But there does appear to be a difference, in that CHI is knowledge-based and CHI's programs are described in a higher-level description-oriented language. The availability of a very-high-level description provides potential for the use of additional knowledge in program compilation and modification. The hope is that this self description and the knowledge base can lead to a set of programming tools that are more powerful not only for creating target end-user programs but also for extending the programming environment. Extending the environment can in turn further facilitate modifying target end-user programs. An example is in a new application where the program editor or pretty printer (part of the environment) must be extended to edit or print some new information structure (part of the target program). The tools provided by the programming environment can more easily assist in this modification process if the environment is itself described in terms that the modification tools can deal with. We have

shown in this paper the feasibility of describing and implementing the programming environment in the system's own very-high-level language.

A drawback of knowledge-based systems is that the addition of new application-domain rules often slows down system performance. In our case, where the application domain is programming, the new knowledge that is introduced can be utilized to speed CHI up. The speed-up helps to mitigate the slowdown caused by the introduction of new alternatives to consider during program synthesis. The net result may well be that system performance improves as new programming knowledge is added. An example would be that as rules are introduced for implementing sets in some new form, say balanced trees, the new tree data structure would be used where appropriate by CHI to implement sets when CHI recompiles itself. In addition, smart self-compilation allows the possibility that new knowledge can be invoked only at reasonable times so that search time is not increased. Another way the descriptive capability helps as new knowledge is added, is that different pieces of the environment are driven off the same internal representations. For example, when a new rule format is added, the reader, printer, editor and compiler all use the same description of the rule format so that consistency is maintained.

In a knowledge-based programming system it can be difficult to draw a boundary between a program specification and general programming knowledge. For example, the specification of RC contains several rules that are specific to RC, dealing with particular rule formats and names of CHI functions. Yet other rules are simplifications such as removing redundant nesting of conjunctions or optimizations such as moving a test outside the scope of a quantifier where possible. The philosophy of a knowledge-based programming system is that general rules such as these are part of the knowledge base and are available to be used whenever appropriate, and are not part of any particular program. For clarity we have included all the general rules in this paper, but if we wished to claim further economy of expression we could argue that the general rules are not a proper part of RC, and thus the specification is really less than half the size presented here. One reason most of the rules are general is that RC deals with the mapping of declarative logic into procedures; the necessary ideas tend to be rather fundamental to programming and are more plausibly classified as general.

1.1 Related work

Situation variables in predicate calculus are used in this paper to formally state the input-output specifications of the desired target code to be produced by the rule compiler, and are also used to give the semantics of a refinement rule. However, for convenience the high-level notation omits the explicit dependence on situations unless necessary. The method of introducing situation variables into the predicate calculus to describe state changes was first introduced in Green 1968 and expanded in Green 1969a. Current progress in the use of this method is exemplified in Manna and Waldinger 1980.

Other knowledge-based programming systems are exemplified by the TI

system (Balzer 1981), and the Programmer's Apprentice (Rich 1980, Shrobe 1979, and Waters 1978). Very-high-level specification languages are exemplified by GIST (Goldman & Wile, 1979) and SETL (Schonberg *et al.* 1978). GIST is currently compiled interactivly with the TI transformation system and SETL contains some sophisticated compiler optimizations.

A very-high-level self description of part of a compiler to produce programs from a logic specification using situation variables was first given in Green 1969b. A theorem prover was the method used to interpret and compile programs specified in the predicate calculus. The control structure of the theorem prover (the interpreter and compiler) was itself described in predicate calculus. But this engine was not powerful enough to either interpret or compile the program. A logic-based technique was used again, this time with an improved theorem prover and specification language in PROLOG to describe a compiler, and in this case PROLOG was able to interpret the compiler (Warren 1980). But the described program was not part of the PROLOG system itself. In the case of CHI, a system that was self-described using situation calculus was compiled. Another closely related work is that of Weyhrauch (Weyhrauch 1980) in which he describes part of the FOL system in its own logic language. This description can be procedurally interpreted to control reasoning in FOL and aids in extending its area of expertise. MRS (Genesereth and Lenat 1980) also features a framework where self-description is used to control reasoning in the system.

2. SPECIFICATION OF THE RULE COMPILER

The rules we are considering have the form:

$$P \rightarrow Q$$

which loosely means 'if P is true in the current situation then transform the situation to make Q true'. P and Q are conjunctions of predicates involving pattern variables. An example of a rule is:

$$class(a) = \text{set} \wedge element(a) = x \rightarrow class(a) = mapping \wedge domain(a) = x \wedge range(a) = \text{boolean}$$

which transforms a set data structure into a Boolean mapping data structure. This is easier to read in the equivalent form with pattern expressions:

$$a\text{:`}set\ of\ x\text{'} \rightarrow a\text{:`}mapping\ from\ x\ to\ \text{boolean}\text{'}$$

The input/output specification of the rule $P \rightarrow Q$ can be stated formally as:

$$\forall x_1, \ldots, x_n [P(s) \Rightarrow Q(succ(s))] \qquad (*)$$

where x_i are the free variables of P, $P(s)$ means that P is true in situation s, and $succ(s)$ is the situation after the rule is applied to situation s. We assume frame conventions that specify, with certain exceptions, that what is true in the initial state is true in the successor or final state unless the rule implies otherwise.

A predicate can be instantiated with objects from our domain of discourse. Each instantiated predicate corresponds to a relation† being in the database. An instantiated left-hand-side predicate is satisfied if the corresponding relation is in the initial state of the database. An instantiated right-hand-side predicate is satisfied by putting the corresponding relation in the final state of the database. Thus the specification can be satisfied by enumerating all instantiations of the quantified variables and for each instantiation in which all the relations of the left-hand side are in the initial state of the database, adding the relations of the right-hand side to the final state of the database. The main task of the rule compiler is to use constraints in the left-hand side of a rule to limit the number of instantiations that need to be enumerated. This optimization is done on the specifcation itself by reducing quantification.

RC has four main stages. The initial stage constructs the input/output specification (*) of the rule. The second stage applies optimizing transformations to the specification by bounding quantifiers and minimizing their scope. The third stage specifies how to satisfy logic expressions using standard high-level programming constructs. The last stage converts database accesses to implementation-specific constructs. We shall be examining stages 2 and 3 in detail.

In order to make the specification cleaner we have presented a few of the rules in a form which RC cannot compile. In section 3 we show how RC can be extended to compile all the rules presented.

2.1 Guide to rule subset of the V language

The rules are constructed from predicates that correspond to relations in the database. The rules of RC transform logic expressions and program expressions, so we need to show how such expressions are represented as relations in the database. For example, the expression *if* $p(x,y)$ *then* $f(x)$ is represented internally by an object w_1 with the following attribute-value relations:

$$class(w_1) = \text{conditional}, condition(w_1) = w_2, action(w_1) = w_3$$

where w_1 and w_2 are objects with the following properties:

$$class(w_2) = \text{p}, arg_1(w_2) = v_1, arg_2(w_2) = v_2,$$
$$class(w_3) = \text{f}, arg_1(w_3) = v_1,$$
$$class(v_1) = \text{var}, name(v_1) = \text{x},$$
$$class(v_2) = \text{var}, name(v_2) = \text{y}.$$

One rule conjunction which matches this representation is:

$$class(a) = \text{conditional} \wedge condition(a) = P \wedge action(a) = Q$$

† In our system the relation $R: X \times Y$ is stored as a function of the form $f: X \mapsto Y$, where $y = f(x)$ *iff* $R(x,y)$ if R is one to one or many to one, or $f: X \mapsto$ *set of* Y, where $y \in f(x)$ *iff* $R(x,y)$ if R is one to many or many to many. The inverse relation is represented by a function in a similar way. Functions may also be computed.

with instantiations: $a \leftarrow w_1; P \leftarrow w_2; Q \leftarrow w_3$. Because rule conjunctions refer to the representation of the expression to be matched rather than the expression itself, it can be difficult to understand them. Therefore we introduce an alternate notation for rule conjunctions which we call pattern expressions. The pattern expression for the previous conjunction is:

a:'*if* P *then* Q'.

Pattern expressions are useful for the reader to see the form of the expressions that the rule refers to, but it is the conjunctive form that is compiled by RC.

In this paper we follow certain naming conventions for pattern variables. Variables starting with S stand for sets of objects. All other pattern variables stand for individual objects. Thus in the pattern a:'$\forall S[C \Rightarrow P]$, S matches the set of quantified variables of a, whereas in the pattern a:'$\forall x\,[C \Rightarrow P]$', x matches the single quantifier variable of a. The other variable naming conventions do not affect the way the rules are compiled. They are: P, Q, R and C match boolean-valued expressions (C is used specifically for conjunctions); u, v and t match terms; p and q match predicates; f matches functions.

2.2 The rule compiler rules

We first present the rules without examples of their use. Then, in section 2.3, we present the steps in the compilation of a rule, which involves applications of all the rules in the current section. It may be useful for the reader to read these two sections together.

Stage 1: Conversion to input/output specification

Creating the specification of the rule involves determining the quantification of variables. The quantification convention is that variables that appear only on the right-hand side are existentially quantified over the right-hand side and variables on the left-hand side are universally quantified over the entire rule. The reason for variables on the right being existentially quantified is that we want to create new objects in the new situation, and this can be expressed by stating existence.

The rule that does the conversion is:

$$a\text{:'}P \rightarrow Q\text{'} \wedge \mathit{FreeVars}(P) = S_0 \wedge \mathit{FreeVars}(Q) - S_0 = S_1 \rightarrow a\text{:'}\mathit{Satisfy}(\forall S_0[P \Rightarrow \exists S_1[Q]])\text{'} \qquad \text{(SatisfySpec)}$$

where $\mathit{FreeVars}(P)$ is the set of free variables in the expression P except for those declared to be bound global to the rule. Note that we have not specified that P refers to the initial state and Q refers to the succeeding state. This could be done by marking predicates as referring to either the initial state or the succeeding state. It turns out that in the current version of RC this marking is unnecessary because the initial separation of left- and right-hand-side predicates is maintained. At the end of stage 3, predicates that refer to the final state are marked as having to be *Satisfied*. How RC can be extended to take advantage of situational tags is discussed in section 3.2.

Stage 2: Optimizing transformations

The rules in this stage do most of the optimizations currently in RC, using equivalence transformations. We do this within logic rather than on a procedural form of the rule because there is a well-understood repertory of logical equivalence transformations. The choice of equivalence transformations is made with a procedural interpretation in mind. The effect of these transformations is to explicate constraints on the evaluation order of the left-hand-side conjuncts. This reflects in the logic form as dependencies being explicit, for example an expression being outside a quantification rather than unnecessarily inside it.

To give an idea of the overall effect of stages 2 and 3, we show the compilation of the rule **SubstBind** before and after these stages. Its purpose is not important here.

$$a\text{:`}Satisfy(P/S_0)\text{'} \wedge y \in S_0 \wedge y\text{:`}x/t\text{'} \rightarrow$$
$$a\text{:`}bind\ S_1\ do\ Satisfy(P)\text{'} \wedge z \in S_1 \wedge z\text{:`}z \leftarrow t\text{'} \qquad \text{(SubstBind)}$$

We abbreviate the right-hand side to *RHS* as we are concentrating on the left-hand side. At the beginning of stage 2, the rule has been transformed to:

$$Satisfy(\forall S_0, y, x, t[class(a) = \text{substitute} \wedge satisfy(a) \wedge substexp(a) = P \wedge substset(a) = S_0 \wedge y \in S_0 \wedge class(y) = \text{subst} \wedge var(y) = x \wedge substval(y) = t \Rightarrow RHS]) \qquad (2.1)$$

At the end of stage 2:

$$Satisfy(class(a) = \text{substitute} \wedge satisfy(a) \Rightarrow (\forall y \in S_0[class(y) = \text{subst} \Rightarrow (RHS/\{x/var(y), t/substval(y)\})])/\{P/substexp(a), S_0/substset(a)\}) \qquad (2.2)$$

At the end of stage 3:

```
if Test(class(a) = substitute ∧ satisfy(a)) then
  bind P ← substexp(a), S0 ← substset(a)
    do enumerate y in S0                                    (2.3)
      do if Test(class(y) = subst) then
        bind x ← var(y), t ← substval(y)
          do RHS
```

The rule compiler determines the order and manner in which each of the eight left-hand-side conjuncts of (2.1) is treated. The fate of the conjuncts can be seen in (2.3). The conjuncts $class(a) = \text{substitute}$ and $satisfy(a)$ can be tested immediately because they depend only on the variable a which is bound externally. The conjunct $substexp(a) = P$ requires that the value of the unknown variable P be equal to an expression in the known variable a, so it is selected next and used to compute the value of P. Similarly, the conjunct $substset(a) = S_0$ is used to compute the value of S_0. Of the remaining conjuncts, $y \in S_0$ is selected next because it contains only the unknown y and so can be used to compute the possible values of y, which is done using an enumeration. This leaves $class(y) = \text{subst}$ with no unknown variables so it is used as a test, and $var(y) = x$ and $substval(y) = t$

give known expressions equal to x and t respectively and so are used to compute values for x and t. Briefly, RC turns conjuncts with no unknown variables into tests and conjuncts with one unknown variable into computations to find the possible values(s) of this variable. At present RC cannot handle conjuncts with more than one unknown variable.

These structural changes, which reflect dependencies among the conjuncts, are performed in stage 2 within logic. In stage 3 implications 'become' *if* statements, substitutions 'become' varible bindings, and bounded universal quantifications 'become' enumerations.

Note that for convenience we have given the conjuncts of the rule in the order in which they are used in the compiled rule, but this is not required by RC.

Another view of RC is that it produces a procedure to find the anchored matches of simple graph patterns. Variables are nodes of the graph and conjuncts are links. At any point there are known nodes and unknown nodes. An unknown node becomes known by following a link from a known node. The structure of variable binding in the target program (2.3) corresponds to a spanning tree of the graph pattern. Consider the expression $RHS/\{x/var(y), t/substval(y)\}$ which is matched when **SubstBind** itself is being compiled. a has the whole expression as its value (actually the object representing the expression); P has the value RHS; S_0 has the set $\{x/var(y), t/substval(y)\}$ as its value; y first has the value $x/var(y)$, x the value x, and z the value $var(y)$; y then has the value $t/substval(y)$, x the value t, and z the value $substval(y)$.

2a) *Reducing quantification scope*

The following rule, when applicable, moves expressions outside the scope of a quantification. If the quantification later becomes an enumeration, the evaluation will be done outside the enumeration instead of inside. The equivalence can be loosely stated: if p is not dependent on x then $(\forall x[p \wedge q \Rightarrow r]) \equiv (p \Rightarrow \forall x[q \Rightarrow r])$.

The corresponding rule expression of this is†:

$$\begin{aligned} a{:}\,&`\forall S[C_0 \Rightarrow Q]' \wedge P \in conjuncts(C_0) \wedge NoVarsOf(P,S) \\ &\rightarrow a{:}\,`C_1 \Rightarrow \forall S[C_0 \Rightarrow Q]' \wedge class(C_1) = \text{conjunction} \\ &\quad \wedge P \in conjuncts(C_1) \wedge P \notin conjuncts(C_0) \qquad \text{(ReduceScope)} \end{aligned}$$

where $NoVarsOf(P,S)$ is true when the expression P is not a function of any of the variables S. Note that there may be more than one conjunct P for which the left-hand side is true, in which case C_1 will have more than one conjunct. Because of the later procedural interpretation of implication (**ImpIIf**) the conjuncts added to C_1 will be tested before those remaining in C_0. However, there is no necessary ordering among the conjuncts of C_1. Note also that we want C_1 to have *only* those conjuncts P such that $P \in conjuncts(C_0) \wedge NoVarsOf(P,S)$, but this is

† We use the convention that the antecedent of an implication is always a conjunction, possibly with only one conjunct.

not explicitly stated. It is implicit that the compiled rule produces the minimal situation that satisfies the specification.

2b) Bounding quantifiers

The following two rules recognize explicit bounds on quantifiers and move the quantification of these variables outside the quantification of any as-yet-unbounded quantifiers. This often enables these rules (and also **ReduceScope**) to be applicable to the inner quantification. This splitting explicates an implicit dependency of the internal quantification on the external quantifiers.

The first rule uses the idea of the following logical equivalence:

$$\forall x[(x \in S \wedge p) \Rightarrow q] \equiv \forall x \in S[p \Rightarrow q]$$

The actual rule is more complicated mainly because there may be more than one quantifier, and the bounded quantification is separated and moved outside any remaining unbound quantifiers.

This allows the inner quantified expression to be manipulated independently.

$$y\text{:`}\forall S_0[C \Rightarrow Q]\text{'} \wedge a \in conjuncts(C) \wedge a\text{:`}x \in t\text{'} \wedge x \in S_0 \wedge NoVarsOf(t, S_0)$$
$$\rightarrow y\text{:`}\forall S_1[\forall S_0[C \Rightarrow Q]]\text{'} \wedge x \notin S_1 \wedge x \notin S_0 \wedge univset(x) = t$$
$$\wedge a \notin conjuncts(C) \qquad \text{(BoundForall)}$$

where $univset(x) = t$ means that x can only take values in the set given by the term t.

The following rule is a special case where a quantifier can only take on one value because it is asserted to be equal to some term independent of the quantifiers. We express this by stating that the quantifier is substituted by this term in the expression, but we do not actually perform the substitution.

$$a\text{:`}\forall S_0[C \Rightarrow Q]\text{'} \wedge y \in conjuncts(C) \wedge y\text{:`}x = t\text{'} \wedge x \in S_0 \wedge NoVarsOf(t, S_0)$$
$$\rightarrow a\text{:`}\forall S_0[C \Rightarrow Q]/S_1\text{'} \wedge z \in S_1 \wedge z\text{:`}x/t\text{'} \wedge x \notin S_0 \wedge y \notin conjuncts(C)$$
$$\text{(ForallSubst)}$$

Stage 3: Interpreting input/output specification procedurally

In this stage the rule is converted from predicate calculus to procedural language. We assume the initial situation is given and that actions necessary to create the successor situation from the initial situation must be performed so that the rule specification is satisfied. Each rule specifies a high-level procedural form for satisfying a particular logical form.

Implication becomes an *if* statement.

$$a\text{:`}Satisfy(C \Rightarrow R)\text{'} \rightarrow a\text{:`}if\ Test(C)\ then\ Satisfy(R)\text{'} \qquad \text{(ImplIf)}$$

Test(C) is true if C is satisfied in the initial state. *Test* is not explicitly used by any of the following rules, but predicates which are not to be *Satisfied* are to be *Tested*.

The following rule says that 'substitution' is actually done using variable binding rather than substitution.

$$a\text{:`}Satisfy(P/S_0)\text{'} \wedge y \in S_0 \wedge y\text{:`}x/t\text{'}$$
$$\rightarrow a\text{:`}bind\ S_1\ do\ Satisfy(P)\text{'} \wedge z \in S_1 \wedge z\text{:`}x \leftarrow t\text{'} \qquad \text{(SubstBind)}$$

An existential variable appearing in the new situation is handled by creating a new object with the specified properties.

$$a\text{:`}Satisfy(\exists S_0[P])\text{'} \wedge y \in S_0 \rightarrow a\text{:`}bind\ S_1\ do\ Satisfy(P)\text{'} \wedge x \in S_1 \wedge x\text{: `}y \leftarrow (NewObject)\text{'} \quad \text{(ExistBindNew)}$$

A conjunction can be satisfied by satisfying each of the conjuncts. In this specification we assume that they can be satisfied independently.

$$a\text{:`}Satisfy(C)\text{'} \wedge class(C) = \text{conjunction} \wedge P \in conjuncts(C) \rightarrow class(a) = \text{block} \wedge Q \in steps(a) \wedge Q\text{:`}Satisfy(P)\text{'} \quad \text{(AndBlock)}$$

Bounded quantification becomes an enumeration:

$$a\text{:`}Satisfy(\forall x \in S[R])\text{'} \rightarrow a\text{:`}enumerate\ x\ in\ S\ do\ Satisfy(R)\text{'} \quad \text{(ForallEnum)}$$

Stage 4: Refine to standard database access functions

4a) Rules for object-centered database implementation

The following rules convert references to functions into references to the database. The particular database representation we use is that the function value $f(u)$ is stored as the f property of the object u. Objects are thus mappings from function names to values. This arrangement may be thought of either as a distributed representation of functions or as the function being indexed by its argument.

$$a\text{:`}f(u)\text{'} \rightarrow a\text{:`}(GetMap\ u\ f)\text{'} \quad \text{(MakeGetMap)}$$

$$a\text{:`}Satisfy(f(u) = v)\text{'} \rightarrow a\text{:`}(ExtendMap\ u\ f\ v)\text{'} \quad \text{(MakeExtMap)}$$

$$a\text{:`}Satisfy(p(u))\text{'} \rightarrow a\text{:`}(ExtendMap\ u\ p\ True)\text{'} \quad \text{(MakeExtMapT)}$$

Note that we have not made all the preconditions for **MakeGetMap** explicit. It should only be applicable when **MakeExtMap** and **MakeExtMapT** are not applicable.

4b) System-specific transformations

The rules of this section are specific to the conventions we use to implement the abstract database. We only present the two of them that reflect issues particularly relevant to the problem of compiling logic specifications. The others convert accesses to abstract datatypes, such as mappings, into accesses to concrete datatypes. These are part of the standard CHI system.

The first rule is for the case where the class of an object gets changed. There are frame conventions which say which of the other properties of the object are still valid. The function *Ttransform* enforces these conventions at rule execution time.

$$a\text{:`}(ExtendMap\ u\ \text{class}\ v)\text{'} \rightarrow a\text{:`}(Ttransform\ u\ v)\text{'} \quad \text{(MakeTtransform)}$$

Creating a new object in the database requires that its own class be known; the object is created as an instance of its class.

$$a\text{:`}(\mathit{ExtendMap}\ u\ \text{class}\ v)\text{'} \wedge a \in \mathit{steps}(P) \wedge z\text{:`}u \leftarrow (\mathit{NewObject})\text{'} \rightarrow z\text{:`}u \leftarrow (\mathit{Tinstance}\ v)\text{'} \wedge a \notin \mathit{steps}(P) \qquad \text{(MakeTinstance)}$$

Simplification rules

The following are general simplification rules that are needed by RC to canonicalize expressions to ensure that other rules will be applicable when appropriate.

$$\mathit{class}(a) = \text{conjunction} \wedge \mathit{class}(C) = \text{conjunction} \wedge a \in \mathit{conjuncts}(C) \wedge P \in \mathit{conjuncts}(a) \rightarrow P \in \mathit{conjuncts}(C) \wedge a \notin \mathit{conjuncts}(C) \qquad \text{(SimpAndAnd)}$$

$$a\text{:`}C \Rightarrow Q\text{'} \wedge \mathit{Null}(\mathit{conjuncts}(C)) \rightarrow \mathit{Replace}(a, Q) \qquad \text{(SimpImpl)}$$

($\mathit{Replace}(a,P)$ causes a to be replaced by P in the expression tree. Formalizing *Replace* is beyond the scope of this paper.)

$$a\text{:`}\forall S[P]\text{'} \wedge \mathit{Null}(S) \rightarrow \mathit{Replace}(a,P) \qquad \text{(SimpForall)}$$

2.3 Sample rule compilation

We present the steps in compiling a representative rule, **ExistBindNew**:

$$a\text{:`}\mathit{Satisfy}(\exists S_0[P])\text{'} \wedge y \in S_0 \rightarrow a\text{:`}\mathit{bind}\ S_1\ \mathit{do}\ \mathit{Satisfy}(P)\text{'} \wedge x \in S_1 \wedge x\text{:`}y \leftarrow (\mathit{NewObject})\text{'}$$

Replacing pattern expression by conjunctions:†

$$\begin{aligned}\mathit{class}(a) = \text{exists} &\wedge \mathit{satisfy}(a) \wedge \mathit{quantifiers}(a) = S_0 \\ &\wedge \mathit{matrix}(a) = P \wedge y \in S_0 \\ \rightarrow \mathit{class}(a) = \text{bind} &\wedge x \in \mathit{bindings}(a) \wedge \mathit{body}(a) = P \\ &\wedge \mathit{satisfy}(P) \wedge \mathit{class}(x) = \text{binding} \wedge \mathit{var}(x) = y \\ &\wedge \mathit{initval}(x) = e_1 \wedge \mathit{class}(e_1) = \text{NewObject}]])\end{aligned}$$

We now apply **SatisfySpec** assuming that a is given as a parameter to the rule so is not quantified within the rule:

$$\begin{aligned}\mathit{Satisfy}(\forall S_0, P, y\,[&\mathit{class}(a) = \text{exists} \wedge \mathit{satisfy}(a) \wedge \mathit{quantifiers}(a) = S_0 \\ &\wedge \mathit{matrix}(a) = P \wedge y \in S_0 \\ \Rightarrow \exists x, e_1\,[&\mathit{class}(a) = \text{bind} \wedge x \in \mathit{bindings}(a) \\ &\wedge \mathit{body}(a) = P \wedge \mathit{satisfy}(P) \\ &\wedge \mathit{class}(x) = \text{binding} \wedge \mathit{var}(x) = y \\ &\wedge \mathit{initval}(x) = e_1 \wedge \mathit{class}(e_1) = \text{NewObject}]])\end{aligned}$$

Note that *Satisfy* is distinct from *satisfy*. Their relationship is not important for purposes of this example.

We next apply the stage 2 rules to the specification to reduce quantification. For now we abbreviate the existential expression to *RHS* as it does not influence anything for awhile, and concentrate on the universal quantification.

† We have also eliminated the unnecessary variable S_1 by replacing $(\mathit{bindings}(a) = S_1 \wedge x \in S_1)$ by $x \in \mathit{bindings}(a)$.

Apply **ReduceScope** to move $class(a) = \text{exists}$ and $satisfy(a)$ outside the quantification:

$$class(a) = \text{exists} \wedge satisfy(a) \Rightarrow \forall S_0, P, y\,[quantifiers(a) = S_0 \wedge matrix(a) = P \wedge y \in S_0 \Rightarrow RHS]$$

Apply **ForallSubst** to fix values for S_0 and P in terms of a:

$$class(a) = \text{exists} \wedge satisfy(a) \Rightarrow (\forall y\,[y \in S_0 \Rightarrow RHS]/\{S_0/quantifiers(a), P/matrix(a)\})$$

Apply **BoundForall** to bound the remaining universal quantifier and simplify away the inner implication with **SimpImpl**:

$$class(a) = \text{exists} \wedge satisfy(a) \Rightarrow (\forall y \in S_0\,[RHS]/\{S_0/quantifiers(a), P/matrix(a)\})$$

Bring back RHS and focus on the universal quantification expression:

$$(\forall y \in S_0)(\exists x, e_1)[class(a) = \text{bind} \wedge body(a) = P \wedge satisfy(P) \wedge x \in bindings(a) \wedge class(x) = \text{binding} \wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}]$$

Moving expressions not depending on y, x and e_1 outside the quantifications (the rules that do this have not been shown before and will be defined in section 3.2):

$$class(a) = \text{bind} \wedge body(a) = P \wedge satisfy(P) \wedge (\forall y \in S_0)(\exists x, e_1)[x \in bindings(a) \wedge class(x) = \text{binding} \wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}]$$

This brings us to the end of stage 2 rules:

(Note that we could have interleaved the following applications of stage 3 rules with those of stage 2 without affecting the final outcome.) The complete rule is:

$$Satisfy(class(a) = \text{exists} \wedge satisfy(a) \Rightarrow (class(a) = \text{bind} \wedge body(a) = P \wedge satisfy(P) \wedge (\forall y \in S_0)(\exists x, e_1)[x \in bindings(a) \wedge class(x) = \text{binding} \wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}])/\{S_0/quantifiers(a), P/matrix(a)\})$$

Applying **ImpIIf**:

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**

$$Satisfy((class(a) = \text{bind} \wedge body(a) = P \wedge satisfy(P) \wedge (\forall y \in S_0)(\exists x, e_1)[x \in bindings(a) \wedge class(x) = \text{binding} \wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}])/\{S_0/quantifiers(a), P/matrix(a)\})$$

Applying **SubstBind**:

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**
  (**bind** $S_0 \leftarrow quantifiers(a), P \leftarrow matrix(a)$
    do $Satisfy(class(a) = \text{bind} \wedge body(a) = P \wedge satisfy(P)$
      $\wedge (\forall y \in S_0)(\exists x, e_1)[x \in bindings(a) \wedge class(x) = \text{binding}$
        $\wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}])$

Applying **AndBlock**:

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**
  (**bind** $S_0 \leftarrow quantifiers(a), P \leftarrow matrix(a)$
    do $Satisfy(class(a) = \text{bind})$
      $Satisfy(body(a) = P)$
      $Satisfy(satisfy(P))$
      $Satisfy((\forall y \in S_0)(\exists x, e_1)[x \in bindings(a) \wedge class(x) = \text{binding}$
        $\wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}])$

Applying **ForallEnum** to the last *Satisfy* expression:

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**
  (**bind** $S_0 \leftarrow quantifiers(a), P \leftarrow matrix(a)$
    do $Satisfy(class(a) = \text{bind})$
      $Satisfy(body(a) = P)$
      $Satisfy(satisfy(P))$
      enumerate y **in** S_0
        do $Satisfy(\exists x, e_1[x \in bindings(a) \wedge class(x) = \text{binding}$
          $\wedge var(x) = y \wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject}])$

Applying **ExistBindNew** to the last *Satisfy* (it is here that we are applying the rule to part of itself):

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**
  (**bind** $S_0 \leftarrow quantifiers(a), P \leftarrow matrix(a)$
    do $Satisfy(class(a) = \text{bind})$
      $Satisfy(body(a) = P)$
      $Satisfy(satisfy(P))$
      enumerate y **in** S_0
        do (**bind** $x \leftarrow (NewObject), e_1 \leftarrow (NewObject)$
          do $Satisfy(x \in bindings(a) \wedge class(x) = \text{binding} \wedge var(x) = y$
            $\wedge initval(x) = e_1 \wedge class(e_1) = \text{NewObject})))$

Applying **AndBlock** to the last *Satisfy*:

if $Test(class(a) = \text{exists} \wedge satisfy(a))$ **then**
  (**bind** $S_0 \leftarrow quantifiers(a), P \leftarrow matrix(a)$
    do $Satisfy(class(a) = \text{bind})$
      $Satisfy(body(a) = P)$
      $Satisfy(satisfy(P))$
      enumerate y **in** S_0

```
do (bind x ← (NewObject), e1 ← (NewObject)
    do Satisfy (x ∈ bindings (a))
       Satisfy (class (x) = binding)
       Satisfy (var (x) = y)
       Satisfy (initval (x) = e1)
       Satisfy (class (e1) = NewObject)))
```

This is the end of stage 3. After applying the stage 4 rules we get the following program which is LISP code except for certain function names and minor syntactical differences.

```
(if (GetMap a 'class) = 'exists and (GetMap a 'satisfy) then
  (bind S0 ← (GetMap a 'quantifiers), P ← (GetMap a 'matrix)
    do (Ttransform a 'bind)
       (ExtendMap a 'body P)
       (ExtendMap P 'satisfy True)
       (for y in S0
          do (bind x ← (Tinstance 'binding),
                   e1 ← (Tinstance 'NewObject)
             do (AddElement x (GetMap a 'bindings))
                (ExtendMap x 'var y)
                (ExtendMap x 'intval e1)))))
```

2.4 Control structure of rule compiler

We have largely succeeded in making the preconditions of the rule compiler rules explicit in the rules themselves, allowing the rules to be used in a wide variety of control structures. The choice of control structure can then concentrate on the issues of efficiency of compilation and efficiency of the target code produced by the compiler. At one extreme the user may interactively control the order of the rule application in order to produce the most efficeint code. At the other, the rules can be incorporated into a program which exploits their interrelations: commonalities in preconditions of rules can be used to produce a decision tree; dependencies between rules, such as the action of one rule enabling the preconditions of others, can be used to direct the order in which rules are tried.

There are some rules whose preconditions we have not made explicit. **MakeGetMap** should not be applied before **MakeExtMap** and **MakeExtMapT.** This constraint could be incorporated explicitly into **MakeGetMap** by adding negations of the distinguishing predicates of **MakeExtMap** and **MakeExtMapT.** The system could embody the general principle that a rule that has a weaker precondition than others should be applied after them, or, alternatively, distinguishing predicates could be added to the weaker rule by the system. This would give the user the choice of specifying the preconditions of a rule implicitly by reference to other rules rather than explicitly within the rule itself.

The control structure we are currently using to automatically compile rules is very simple but provides a reasonable compromise between efficiency of

execution, efficiency of the code produced, and flexibility (changing one rule does not require that the entire rule compiler be recompiled). The expression tree for the rule being compiled is traversed depth-first applying each rule to each object on the way down. If an object is transformed by a rule then the traversal continues from this object. The order in which rules are applied to a particular node affects the efficiency of the code produced. In particular it is desirable to apply **ReduceScope** before **ForallSubst** and **ForallSubst** before **BoundForall.**

3. EXTENSIONS TO THE RULE COMPILER

One of the main tests of our high-level description of RC is how easy it is to extend. In this section we show how it can be extended along various dimensions: improving efficiency of the target code, augmenting the rule language, improving the user interface, and adapting to other system tasks.

3.1 Improving efficiency

We give examples of adding rules in order to improve the execution efficiency of compiled rules. Adding rules will tend to slow down the rule compiler, but as RC is specified primarily in terms of rules, after recompiling its own rules RC may become substantially faster. This section considers a number of different areas where efficiency can be improved.

Frequently it is possible to express the same program in two ways where one way is simpler or more uniform, but compiles into more inefficient code. The trade-off can be circumvented by modifying a compiler so that it translates the simpler form into the more efficient form. For example, the LISP macro facility allows the user to do this to some extent. Adding new rules to RC provides a more general way of doing this in a more convenient language. A simple example is using a more specific access function for a special case of a general access function. The following rule would be useful if an object were stored as a list whose first element is the class of the object:

$$a\text{:`}(GetMap\ u\ \text{'class})\text{'} \rightarrow a\text{:`}(car\ u)\text{'}$$

Such a speed-up should be derived by CHI from knowledge of how objects are stored. That the rule compiler can be extended easily can be exploited by the rest of CHI as well as by the user directly.

A more extensive change to RC would be to use a different implementation for the database. In particular, a less general data structure could be used for representing the rules. If singly-linked lists were used, the compiled rules could no longer follow inverse links. To compensate for this, the control structure surrounding the rules would have to keep more context in free variables for the compiled rules to refer to. Rather than rewriting the rules that previously used inverse links, only the rules that compile uses of inverse relations need be rewritten.

A useful improvement in efficiency could be gained by combining sets of rules into a single function as indicated in section 2.4. There would be relative advantages in doing the combination using the original rule form or with the

forms produced by stage 2. The changes necessary to RC would be modifications of the control structure so that it could take more than one rule and a new set of rules to do the combining. Few changes to existing rules would be necessary.

A special case of combining rules is where one rule always follows another. This is true for a number of pairs of rules in RC: **BoundForall** is followed by **ForallEnum**; **ForallSubst** is followed by **SubstBind.** We could have specified RC with these rules combined at the cost of less clarity, and less generality of the individual rules. The combination of **BoundForall** and **ForallEnum** is:

$$y\text{:`}Satisfy(\forall S_0[C \Rightarrow Q])\text{'} \wedge a \in conjuncts(C) \wedge a\text{:`}x \in t\text{'} \wedge x \in S_0 \wedge NoVarsOf(t,S) \rightarrow a\text{:`}enumerate\ x\ in\ t\ do\ Satisfy(\forall S_0[C \Rightarrow Q])\text{'} \wedge x \notin S_0 \wedge a \notin conjuncts(C)$$

which is comparable in complexity to **BoundForall** alone. However, it blurs the two things which are going on that are distinguished in the individual rules: a logical equivalence and how procedurally to satisfy a universal quantification. Having the two ideas in separate rules means that they can be applied separately elsewhere. However it may well be desirable to compile them together to increase efficiency.

3.2 Extending the rule language

There are two classes of extensions to the rule language. New constructs and abbreviations can be incorporated by adding rules to stage 1 which translate the new constructs into standard logic constructs compilable by the lower part of RC. The second class of extension is the addition of rules to stage 2 and/or stage 3 to increase RC's coverage of logic constructs. An addition of the first class may require additions to lower parts of RC in order that the new pattern be compilable.

The following two rules free the user from having to decide whether it is necessary to use a particular relation or its inverse. They also loosen constraints on which variables may be chosen as parameters to the rule or as enumeration variables. They are required in order to compile a number of the rules without unnecessary enumerations, including **BoundForall** when a is the parameter.

$$y\text{:`}\forall S[C \Rightarrow Q]\text{'} \wedge a \in conjuncts(y) \wedge a\text{:`}f(u) = t\text{'} \wedge NoVarsOf(t,S) \wedge OneToOne(f) \rightarrow a\text{:`}u = f^{-1}(t)\text{'}$$

$$y\text{:`}\forall S[C \Rightarrow Q]\text{'} \wedge a \in conjuncts(y) \wedge a\text{:`}f(u) = t\text{'} \wedge NoVarsOf(t,S) \wedge ManyToOne(f) \rightarrow a\text{:`}u \in f^{-1}(t)\text{'}$$

A number of rules, including **ExistBindNew**, require the following two rules in order to compile correctly. Like **ReduceScope**, they move expressions outside the scope of quantifications.

$$a\text{:`}\exists S[C]\text{'} \wedge class(C) = conjunction \wedge P \in conjuncts(C) \wedge NoVarsOf(P,S) \rightarrow class(a) = \text{conjunction} \wedge P \in conjuncts(a) \wedge y \in conjuncts(a) \wedge y\text{:`}\exists S[C]\text{'} \wedge P \notin conjuncts(C)$$

$$a\text{:`}\forall S[C]\text{'} \wedge class(C) = conjunction \wedge P \in conjuncts(C) \wedge NoVarsOf(P,S) \rightarrow class(a) = \text{conjunction} \wedge P \in conjucts(a) \wedge y \in conjucts(a) \wedge y\text{:`}\forall S[C]\text{'} \wedge P \notin conjuncts(C)$$

The main part of RC involves compiling the satisfaction of an input/output specification. In the remainder of this section we consider extensions to RC that increase the types of acceptable input/output specifications.

By explicitly marking functions as referring to either the initial or succeeding state we get more freedom in how to mix them in the specification and more freedom in how the specification can be manipulated in stage 2. We use the convention that functions or expressions marked with a single prime, as in f', refer to the initial state, and those marked with a double prime refer to the succeeding state, as in f''.

Consider the case where we allow disjunctions in the input/output specificantion. For example, consider the expression *Satisfy*$(A' \Rightarrow B'')$ in the disjunctive form *Satisfy*$(\neg A' \vee B'')$. To get from the latter to an expression similar to that obtained from applying **ImpIIf** to the former expression, one can apply the rule:

$$a\text{:`}\textit{Satisfy}(P \vee Q)\text{'} \rightarrow a\text{:`}\textit{if UnSatisfiable}(P)\ \textit{then Satisfy}(Q)\text{'}$$

giving *if UnSatisfiable*$(\neg A')$ *then Satisfy*(B''). Comparing this with *if Test*(A') *then Satisfy*(B'') we see that *Test*(x) corresponds to *UnSatisfiable*$(\neg x)$. Note that we could also have derived the program *if UnSatisfiable*(B'') *then Satisfy* $(\neg A')$, but if we have no procedures for testing unsatisfiability in the final state or satisfying things in the initial state, then this choice will lead to a dead end.

In general, relaxing the restrictions on the form of the specification requires additional Stage 3 rules rather than changes to existing ones, but the rules may lead to dead ends, so a more sophisticated control structure is necessary. On the other hand, the Stage 4a rules do require the addition of preconditions concerning whether functions refer to the initial or final state.

For example, **MakeExtMap** becomes:

$$a\text{:`}\textit{Satisfy}(f''(u) = v)\text{'} \rightarrow a\text{:`}(\textit{ExtendMap}\ u\ f\ v)\text{'}$$

with restrictions also necessary on u and v if these are allowed to be expressions more general than variables.

RC in its present form can be used to compile rules that derive implications of the current state rather than transform it. The specification for such rules is exactly the same as (*) except that it is not necessary to distinguish the initial and succeeding states. This distinction is not actually used by RC so therefore RC can compile these rules. The rule **MakeTtransform** is not applicable to compiling such implication rules. Its preconditions being true would imply that the implication rule does not reflect a valid implication.

3.3 Improving user interface

The user interface can be improved in a variety of ways. Extending the rule language, discussed above, is one way. Another is to put error detection rules which report problems in the rule or indicate when the rule is beyond the scope of the rule compiler. This is also useful as self-documentation of the rule compiler.

The following rule can be used to detect any unbound quantifers left after the application of stage 2 rules:

$$a: \text{'}Satisfy(\forall S[P])\text{'} \wedge x \in S \wedge \neg univset(x) \rightarrow Error(x, Unbound)$$

In another dimension, RC can be extended to produce target code that is oriented to the user's use of it. The main use of compiled rules outside the rule compiler is in refining high level programs under user guidance. At each decision point the user wants to know which rules are relevant. The system should help in providing such a list. We have added a set of rules to RC in V that extracts part of the left-hand side to use as a filter. The rule language provides a convenient way for the user to express and experiment with heuristics for determining the relevant parts to test.

3.4 Relation to the rest of the system

In this section we discuss specifically how the rule compiler can benefit the system and how the rest of the system can benefit the rule compiler, apart from the primary purpose of the rule compiler of compiling rules.

Some ways in which the rule compiler can benefit the system have already been covered, such as in providing filters to screen irrelevant rules from the user. The primary contribution of the rule compiler is the provision of a useful high-level language that can be used elsewhere. One immediately applicable example is in compiling user database queries. These are exactly like rules except that they usually do not have any parameters and the actions are typically to display the matching objects (although editing could be performed if desired). An example query may be to print out all rules that apply to universal quantifications:

$$y: \text{'}C \rightarrow Q\text{'} \wedge P \in conjuncts(C) \wedge P: \text{'}class(x) = \text{forall'} \rightarrow Display(y)$$

Improvements in the rest of the system can have benefits for the rule compiler. As mentioned above, the whole program synthesis system may be brought to bear on compiling an important rule. General efficiency knowledge for determining which of several implementations is most efficient would carry over to rule compilation. Also, additions made to take advantage of dataflow or for manipulating enumerations could be applicable to the rule compiler. All the tools for maintaining programs written in V are applicable to maintaining the rule compiler program, including editors, consistency maintenance systems and the system for answering user queries about programs.

Acknowledgements

We would like to acknowledge J. Phillips for numerous key ideas in the design of CHI and V and for suggestions for describing the rule compiler in V. S. Angebranndt helped implement and debug RC in V. T. Pressburger developed an implementation-independent description of CHI's knowledge base. S. Tappel helped to define V and wrote the original version of RC in LISP. R. Floyd and

B. Mont-Reynaud provided considerable technical and editing assistance with this paper.

This work describes research done in Kestrel Institute and Systems Control Inc. This research is supported in part by the Defense Advanced Research Projects Agency Contracts N00014-79-C-0127 and N00014-81-C-0582 monitored by the Office of Naval Research. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Kestrel, SCI, DARPA, ONR or the U.S. Government.

REFERENCES

Balzer, R. (1981). Transformational implementation: an example, *IEEE Transactions on Software Engineering*, Jan. 1981, 3–14.

Genesereth, M. R. and Lenat, D. B. (1980). "A modifiable representation system,' *HPP-80-26*, Stanford: Computer Science Department, Stanford University.

Goldman, N. M., and Wile, D. S. (1979). A relational data base foundation for process specifications, *Int. Conf. on Entity-Relationship Approach to Systems Analysis and Design*, pp. 413–432 (ed. Chem., pp).

Green, C. C. (1969). Theorem proving by resolution as a basis for question answering systems, *Machine Intelligence 4*, pp. 183–205, (eds. Meltzer, B. and Michie, D.). Edinburgh: Edinburgh University Press.

Green, C. C. (1969a). Application of theorem-proving to problem-solving, *Proc. Int. Jnt. Conf. Art. Int.* (LJCAI-69), pp. 219–239, eds. Walker, D. A. and Morton, L. M. London and New York: Gordon and Breach.

Green, C. C. (1969). The application of theorem-proving to question-answering systems, Ph.D. Thesis, Electrical Engineering Department, Stanford University. Also printed as *AIM-96*, and *STAN-CS-69-138*. Stanford: Artificial Intelligence Laboratory, Computer Science Department, Stanford University, also reprinted 1979. New York: Garland Publishing, Inc.

Green, C. C., Phillips, J., Westfold, S., Pressburger, T., Angebranndt, S., Kedzierski, B., Mont-Reynaud, B., and Chapiro (1981). Progress in knowledge-based programming and algorithm design, *Technical Report KES.U.81.1*, Palo Alto: Kestrel Institute.

Ingalls, D. H. (1978). The SMALLTALK-76 programming system: design and implementation, *Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 9–16. Tucson, Arzona, January, 1978.

Kant, E. (1979). Efficiency considerations in program synthesis: A knowledge based approach, Ph.D. Thesis. Also published as *AIM-331*, and *STAN-CS-79-755*. Stanford: Computer Science Department, Stanford University; also as *Efficiency in program synthesis.* Ann Arbour: UMI Research Press (1981).

Kowalski, R. (1979). *Logic for Programming*, Amsterdam, New York, Oxford: North Holland.

Manna, Z., and Waldinger, R. (1980). Problematic features of programming languages: a situational calculus approach, *STAN-CS-80-779*. Stanford: Department of Computer Science, Stanford University.

Phillips, J. P. Self-described programming environments: an application of a theory of design to programming systems. Ph.D. Thesis. Stanford. Departments of Electrical Engineering and Computer Science, Stanford University.

Phillips, J., and Green, C. C. (1980). Towards self-described programming environments, *Technical Report, SCI.ICS.L.81.3*, Palo Alto: Computer Science Department, Systems Control Inc.

Rich, C. (1981). Inspection methods of programming, Ph.D. Thesis, *MIT/AI/TR-604*, Cambridge, Mass: MIT.

Schonberg, J., Schwartz, J. T. and Sharir, M. (1978). Automatic data selection in SETL, *Proc. Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, Jan. 1978.

Shrobe, H., Dependency directed reasoning for complex program understanding, Ph.D. Thesis, *MIT/AIM/TR-503*, Cambridge, Mass: MIT.

Teitelman, W., and Masinter, L. (1981). The INTERLISP programming environment, *Computer*, 14, 4.

Warren, D. H. D., Pereira, L. and Pereira, F. (1977). PROLOG: the language and its implementaion compared with LISP, *Proc. Symposium and AI and Programming Languages*, *SIGPLAN/SIGART*, 12, No. 8.

Warren, D. H. D. (1980). Logic programming and compiler writing, *Software – Practice and Experience*, 97–125.

Waters, W. (1978). Automatic analysis of the logical structure of programs, Ph.D. Thesis, *MIT/AI/TR-492*, Cambridge, Mass: MIT.

Weyhrauch, R. W. (1980). Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* 13, 133–170.