Report 84-01
Stanford -- KSL

Partial Programs.
Michael R. Genesereth,
Nov 1984

Heuristic Programming Project
Report HPP-84-1

First Draft January 1984
Current Version November 1984

Partial Programs

Michael R. Genesereth

Computer Science Department
Stanford University
Stanford, California 94305

Abstract: A partial program is an arbitrary set of constraints on the potential actions of a machine. The advantage of having a language for writing partial programs is ease of incremental program development. In traditional programming languages, arbitrary decisions must often be made in writing runnable programs. In order to change these decisions, it's necessary to rewrite the programs. A partial programming language allows one to write programs without making arbitrary decisions and thereby facilitates the implementation of subsequent design refinements. This paper presents a theory of partial programs and discusses some of their properties. It introduces a particular partial programming language called PRO and describes its interpreter. Finally, it shows how the language can support a variety of programming styles.

1 Introduction

A complete program is one that for any environment specifies a unique action for a machine to perform. Programs in most traditional programming languages are complete in this sense. By contrast, a partial program is merely an arbitrary set of constraints on the potential actions of a machine and does not necessarily specify a unique action in every environment. (Nondeterministic programs are similar in this regard; however, as will be discussed later, traditional nondeterminism is only one way in which a program can be partial.)

The advantage of having a language for writing partial programs is that it facilitates incremental program development. In traditional programming languages, arbitrary decisions must often be made in writing runnable programs. In order to change these decisions, it is necessary to rewrite the programs. A partial programming language allows one to write programs without making arbitrary decisions and thereby facilitates the implementation of subsequent design refinements.

As an example, consider the problem of finding a path between nodes in a directed acyclic graph like the one shown in figure 1.

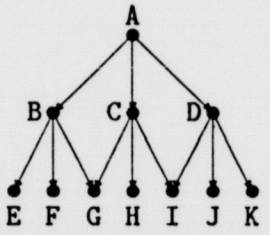

Figure 1: A very simple directed acyclic graph

Figure 2 contains a partial program to solve this problem, written here in English. In the initial state, the only legal action is to mark node A. Once A is marked, it's okay to mark node B or node C or node D; there's no constraint on which should be processed first. If C is marked at this point, it's still okay to mark node B or node D, but it's also okay to mark node

G or node H or node I. Once node J is marked, it's okay to succeed. Since nodes are marked only if they have incoming arcs from marked nodes, the program will succeed only if there is a path from node A to node J.

Mark node A.
It's okay to mark any node with an incoming arc from a marked node.
Succeed if node J is ever marked.

Figure 2: A partial path finding program

Figure 3 contains a complete program consistent with the constraints in figure 2. The difference is that this program dictates a particular search strategy (viz. depth-first search) and a particular order in which to mark the subnodes of a marked node (viz. the order returned by the subroutine CONNECTIONS), whereas the partial program leaves those choices open.

```
PATHP(X,Z) :=
    IF X=Z THEN RETURN(TRUE)
           ELSE FORONE(Y,CONNECTIONS(X),PATHP(Y,Z))
```

Figure 3: A complete path finding program

One can add further constraints to a partial program, like those shown in figure 4. The first constraint merely ensures that, once a node is marked, it will not be marked again. The second constraint assumes the existence of heuristic information to guide the search in the form of a "quality" measure on nodes. Given this heuristic information, the constraint dictates that the unmarked nodes be processed in order of decreasing quality. The final constraint adds a hint of opportunism to the program. There is no guarantee that the program will find any nodes of zero quality, even if they exist; however, if it does encounter one, it will print an appropriate remark. By adding enough constraints a partial program can become a complete program. In this case, the additional propositions are sufficiently constraining that only one action is possible at each point in time (assuming that every node has a different quality).

Do not mark a node that has already been marked.
Mark nodes in order of highest quality.
If a node with zero quality is encountered, say so and continue.

Figure 4: Additional constraints for the program in figure 2

By contrast, in order to incorporate these additional constraints into an already complete program like the one in figure 3, it is necessary to rewrite the code, e.g. as shown in figure 5. In this case the change is fairly simple, but for complex programs the process of identifying arbitrary design decisions and rewriting the code can be tedious or even prohibitive.

This paper is an introduction to partial programs. Section 2 introduces the theory of partial programs and defines a particular partial programming language called PRO. Sections 3, 4, and 5 illustrate three distinct styles of programming using PRO. Section 6 presents an interpreter for the language. The conclusion describes the advantages and disadvantages of partial programs, mentions some directions for future work, and summarizes the key points of the paper.

2 Procedures and Programs

The theory of programs used in this paper assumes that the universe of discourse contains a set of distinguished objects called *states* (also *worlds*, *situations*, etc.). Each state is characterized by the facts true in that state. Two states are distinct if and only if there is some fact true in one state that is not true in the other. In most applications the number of distinct states is infinite.

For reasons that will become clear later, it's useful to divide this set of states into a set of mutually exclusive subsets. Of course, this can be done in many ways. Each such *partitioning* of the set of states can be defined as a function from states to the partitions to which they belong.

An *action* is a state transformation. While it is possible to formalize this notion as a function from states into states, this complicates the representation of some information about actions. An alternative that solves this problem is to reify actions as objects in the universe of discourse and

```
PATHP(X,Z) :=
    BEGIN NODES:[X],
          REPEAT X:BEST(NODES),
                 IF QUALITY(X)=0 THEN PRINT(X," is zero."),
                 IF X=Z THEN RETURN(TRUE),
                 NODES:APPEND(CONNECTIONS(X),DELETE(X,NODES))
          END
    END

BEST(L) :=
    BEGIN BEST:FIRST(X),
          REPEAT L:REST(L),
                 IF L=[] THEN RETURN(BEST),
                 IF GREATERP(QUALITY(FIRST(L)),QUALITY(BEST))
                    THEN BEST:FIRST(L)
          END
    END
```

Figure 5: A traditional version of the partial program in figures 2 and 4

to relate these objects to changes in state via a single function for all actions. In what follows, it is assumed that there is such a function, called *Do*, which maps an action k and a state s into the state resulting from the application of k to s.

A *machine* consists of a partition function on states and a binary relation between states and actions. Some machines are unable to distinguish some states from others. The partition function divides the set of all states into a set of distinguishable subsets. Some machines are unable to perform every action. The binary relation is true of a state s and an action k if the machine is capable of executing action k in state s. A machine is *finite* if and only if it has a finite number of distinguishable partitions and a finite number of possible actions.

As an example, consider a Turing machine with a finite set of *internal* states and an infinite tape with a single head. The overall state of the world is determined by the current internal state of the machine, the current contents of the tape, and the current position of the head. Since there are infinitely many squares on the tape, there are infinitely many overall states. Nevertheless, these states can be partitioned into a *finite* number of categories on the basis of the internal state of the machine and the contents of the tape square under the machine's head. There are also finitely many actions, each of which specifies an internal state to enter, a bit to write on the square under the head, and a movement of the head one square to the left or one square to the right.

A machine like this one is capable of performing any one of several actions in any given state. A procedure is a prescription for which action it should perform. Formally, a *deterministic procedure* is a function F from states to actions. F is legal for a machine with state partition function P and with action relation M if and only if the following conditions hold for every state s.

(1) The machine is capable of executing the prescribed action.

$$F(s) = k \Rightarrow M(s, k)$$

(2) The prescribed action is the same for any two members of the same partition.

$$P(s) = P(t) \Rightarrow F(s) = F(t)$$

As a simple example, consider a procedure of reversing the bits on the tape of the Turing machine described above. If the square under the head contains a one, the procedure prescribes that the machine change it to a zero and move right. If the square contains a zero, the procedure describes that the machine write a one and move right. For a problem this simple no internal state is necessary. This procedure is deterministic since it prescribes a unique action in each state, and it's legal since it satisfies the preceding conditions. Note, however, that it never "terminates".

A nondeterministic procedure is less specific than a deterministic procedure. In any given state more than one action may be appropriate. This can be formalized by defining a *nondeterministic procedure* as a relation between states and actions, rather than a function from states to actions. The conditions for the legality of nondeterministic procedures are similar to those for deterministic procedures.

(1) The machine must be able to execute any action allowed by R.

$$R(s,k) \Rightarrow M(s,k)$$

(2) The allowed actions are the same for any two members of the same partition.

$$P(s) = P(t) \Rightarrow (R(s,k) \Leftrightarrow R(t,k))$$

Informally, a program is a description of a procedure in a programming language. A program is deterministic if it describes a deterministic procedure. A program is nondeterministic if it describes a nondeterministic procedure. In this paper programs are written in a programming language called PRO.

The syntax of PRO is the same as that of first order predicate calculus. Variables begin with lower case letters, and all free variables are universally quantified. Object, function, and relation constants begin with upper case letters, digits, or non-alphabetic characters. Sets are delimited by braces, and sequences are delimited by angle brackets.

The vocabulary of the language includes the usual logical operators (i.e. $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftarrow$) and quantifiers (i.e. $\forall$ and $\exists$). In addition, there are terms to designate the partitions and actions of the machine on which a procedure to be executed.

Formally, a *partial deterministic program* is a tuple $\langle \pi, \Gamma \rangle$ where π is a unary function constant and Γ is any consistent set of PRO propositions (which presumably constrain π in some way). A *partial nondeterministic program* is a tuple $\langle \rho, \Delta \rangle$ where π is a binary relation constant and Δ is any consistent set of PRO propositions.

Note the difference in style between programs in traditional languages and programs written in PRO. A traditional program is a term in which the function constant (e.g. IF or REPEAT) names a particular control primitive (e.g. conditionality or iteration) and the arguments describe the actions controlled by it. A program in PRO is a function or relation symbol and an arbitrary set of constraints. Examples are given in the next three sections. The following discussion summarizes some of the properties of partial programs in general and PRO programs in particular. It can be skipped on first reading without severe loss of continuity.

It is important to realize that the only constraint on the propositions comprising a partial program is consistency. A programmer may describe a procedure in as much or as little detail as he desires. If the description includes too little information, there may be more than one procedure that satisfies the propositions. The extension $|\langle \pi, \Gamma \rangle|$ of a partial deterministic program $\langle \pi, \Gamma \rangle$ is the set of all procedures F for which there is an interpretation I such that $I(\pi) = F$ and $\models_I \Gamma$. The extension of a partial nondeterministic program $\langle \rho, \Gamma \rangle$ is the set of all deterministic procedures F for which these is an interpretation I such that $I(\rho) = H$, $\models_I \Gamma$, and $F(s) = k \Rightarrow H(s,k)$ (i.e. in each state the function selects one of the actions allowed by H).

One can refine a partial program by augmenting it with additional propositions. One can combine programs by forming the union of the propositions comprising the separate programs. The results of such additions and combinations have the following satisfying property.

Theorem 1 *If* $\langle \pi, \Gamma_1 \rangle$ *and* $\langle \pi, \Gamma_2 \rangle$ *are programs, then*

$$|\langle \pi, \Gamma_1 \cup \Gamma_2 \rangle| \subseteq |\langle \pi, \Gamma_1 \rangle| \cap |\langle \pi, \Gamma_2 \rangle|.$$

Proof: If any interpretation I satisfies all of the propositions in $\Gamma_1 \cup \Gamma_2$, then I satisfies all of the propositions in Γ_1 and all of the propositions in

Γ_2. Hence, any procedure determined by an interpretation of $\Gamma_1 \cup \Gamma_2$ is in the extensions of both $\langle \pi, \Gamma_1 \rangle$ and $\langle \pi, \Gamma_2 \rangle$. Note that, if Γ_1 and Γ_2 are inconsistent with each other, then the set of interpretations for $\Gamma_1 \cup \Gamma_2$ is empty. □

In some cases the containment is proper.

Theorem 2 *There are programs* $\langle \pi, \Gamma_1 \rangle$ *and* $\langle \pi, \Gamma_2 \rangle$ *such that*

$$|\langle \pi, \Gamma_1 \cup \Gamma_2 \rangle| \subset |\langle \pi, \Gamma_1 \rangle| \cap |\langle \pi, \Gamma_2 \rangle|.$$

Proof: Consider the programs $\langle \mathtt{F}, \{\mathtt{P}, \mathtt{P} \wedge \mathtt{Q} \Leftrightarrow \mathtt{F(A)} = \mathtt{J}\}\rangle$ and $\langle \mathtt{F}, \{\mathtt{Q}, \mathtt{P} \wedge \mathtt{Q} \Leftrightarrow \mathtt{F(A)} = \mathtt{J}\}\rangle$. The first has an interpretation in which Q is false; and, therefore, any procedure consistent with F(A)≠J is in its extension. The second program has an interpretation in which P is false; and, therefore, any procedure consistent with F(A)≠J is in its extension as well. Consequently, any such procedure is in the intersection of the two extensions. However, F(A)=J is true in every interpretation of the joint program $\langle \mathtt{F}, \{\mathtt{P}, \mathtt{Q}, \mathtt{P} \wedge \mathtt{Q} \Leftrightarrow \mathtt{F(A)} = \mathtt{J}\}\rangle$; and, therefore, its extension does not contain any procedure consistent with F(A)≠J. □

The significance of this result is that one must be careful in combining partial programs that the assumptions in writing one program and the assumptions in writing the other program don't combine to eliminate procedures intended to be in the extension of th joint program. Interactions of this sort can be caught so long as the assumptions are explicitly recorded. However, if the assumptions are left implicit as in the examples in the proof, the contradiction can go unnoticed and an intended procedure can be inadvertently lost.

A program $\langle \pi, \Gamma \rangle$ is *complete* when every model of Γ is homomorphic with respect to π, i.e. for any two interpretations I and J that satisfy Γ, there is a homomorphism h between the universes of discourse of I and J such that for every term σ it's true that $h(\pi^I(\sigma^I)) = \pi^J(h(\sigma^I))$. Intuitively, this means that all interpretations of the program have the same "structure". Two interpretations can disagree on the universe of discourse, but if the elements of one universe are put into correspondence with the elements of the other universe, the actions prescribed by the first interpretation correspond to those prescribed by the other interpretation.

It's easy to show that if a program $\langle \pi, \Gamma \rangle$ is complete according to this definition, then for all terms σ and κ in the language it is either the case that $\Gamma \models \pi(\sigma) = \kappa$ or $\Gamma \models \neg\pi(\sigma) = \kappa$. In other words, a program is complete if it dictates a specific action in each state.

Complete nondeterministic programs and partial deterministic programs are similar in that both describe sets of deterministic procedures. As one might expect, a nondeterministic program is a special kind of partial deterministic program.

Theorem 3 *For every complete nondeterministic program, there is a partial deterministic program with the same extension.*

Proof: Given a complete nondeterministic program $\langle \rho, \Delta \rangle$, one can construct an equivalent partial deterministic program by selecting a new function symbol π and a set Γ of propositions consisting of the elements of Δ together with the new proposition $\forall s \forall k \ \neg\rho(s,k) \Rightarrow \pi(s) \neq k$. The partial deterministic program $\langle \pi, \Gamma \rangle$ then has the same extension as $\langle \rho, \Delta \rangle$. □

The converse, however, is not true. There are partial deterministic programs with no corresponding complete nondeterministic programs.

Theorem 4 *There is a partial program for which there is no complete nondeterministic program with the same extension.*

Proof: As an example, consider an application area in which there are just two states, named A and B, and two actions, named J and K. The partial deterministic program $\langle F, \{F(s) = F(t)\} \rangle$ has an interpretation in which F(A)=J and F(B)=J are true and another interpretation in which F(A)=K and F(B)=K are true. In order for the extension of a nondeterministic program $\langle H, \Delta \rangle$ to admit these two interpretations, H(A,J) and H(B,J) must be true and so must H(A,K) and H(B,K). But then there is a deterministic procedure F in the extension of $\langle H, \Delta \rangle$ for which F(A)=J and F(B)=K are both true. □

The difference here is subtle but important. The ambiguity in a complete nondeterministic program is intentional; it is, in a sense, a *stated lack of preference* for one action or order of actions over another. By contrast, the ambiguity in a partial program is a *lack of stated preference.* Much

research in computer science has been concerned with the invention of control primitives for expressing nondeterminism. This theorem indicates, no matter how flexible one's primitives are, there is always some flexibility that cannot be captured.

3 Simple Example - State Search

The simplest examples of partial programs are those in which each state is completely described by a single term in the PRO language.

A good example of this is the 8-puzzle problem pictured in figure 6. Starting from a state in which the tiles are scrambled as on the left, the goal is to achieve a state in which the tiles are arranged as on the right. The only allowable state transitions are movements of the blank tile up, down, left, or right.

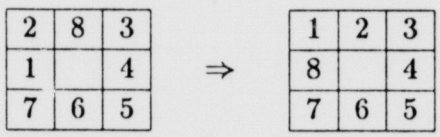

Figure 6: The 8-puzzle

The states for this problem can be represented by 9-tuples. Each component in a 9-tuple represents one space in the puzzle, and the term in that component designates the tile occupying that space. For example, the following tuple represents the state of the 8-puzzle on the left of figure 6.

$$< 2, 8, 3, 1, B, 4, 7, 6, 5 >$$

It's usual to assume that the machine for manipulating an 8-puzzle is sufficiently discriminating that it can distinguish every state from every other state. This means that the machine's partition function P maps each state into its own partition. If one equates states with partitions, the partition function becomes the identity map.

```
P(s)=s
```

There are four actions, named Up, Dn, Lt, and Rt. Of course, not every action is executable in every state. For example, it's illegal to move the blank left from a state in which it is against the left boundary of the puzzle. It's illegal to move the blank up in a state in which it is against the upper boundary of the puzzle. The following propositions characterize the set of legal moves.

```
M(< B,x2,x3,x4,x5,x6,x7,x8,x9>,Dn)
M(< B,x2,x3,x4,x5,x6,x7,x8,x9>,Rt)
¬M(< B,x2,x3,x4,x5,x6,x7,x8,x9>,Up)
¬M(< B,x2,x3,x4,x5,x6,x7,x8,x9>,Lt)
          etc.
```

As defined in the last section, a procedure F is legal for this machine if and only if it satisfies the following constraints. Of course, the definition of P given above ensures that the first constraint is trivially satisfied.

```
P(s)=P(t) ⇒ F(s)=F(t)
F(s)=k ⇒ M(s,k)
```

These legality constraints together with the preceding machine descriptions constitute a partial program for the 8-puzzle problem. The program is partial in that more than one action is legal in any state. In fact, the program describes all possible procedures for solving the problem.

Starting with this base set of propositions, one can specify a particular procedure or select a subset of these procedures by writing additional propositions. For example, one can define an evaluation function Q on states and use it to constrain the machine's transitions to those states that maximize this function. Nilsson [Nilsson] suggests a function based on the number of tiles in their desired locations.

```
x1=1 ⇒ Q1(<x1,x2,x3,x4,x5,x6,x7,x8,x9>)=1
x1≠1 ⇒ Q1(<x1,x2,x3,x4,x5,x6,x7,x8,x9>)=0
  .  .  .
x9=5 ⇒ Q9(<x1,x2,x3,x4,x5,x6,x7,x8,x9>)=1
x9≠5 ⇒ Q9(<x1,x2,x3,x4,x5,x6,x7,x8,x9>)=0
Q(s)=Q1(s)+Q2(s)+Q3(s)+Q4(s)+Q5(s)+Q6(s)+Q7(s)+Q8(s)+Q9(s)
Q(s)>Q(Do(k,s)) ⇒ F(s)≠k
```

`Q(Do(j,s))>Q(Do(k,s))` $\Rightarrow$ `F(s)`$\neq$`k`

The next to last proposition here ensures that F never specifies a move to a state in which fewer tiles are in place. The last proposition ensures that a move that results in more tiles being in place is preferred to a move in which fewer tiles are in place.

This example illustrates the technique of "programming by subtraction". One starts out with a completely general program and successively removes possibilities by adding constraints. Ultimately, the program becomes sufficiently constrained that a unique procedure is specified.

4 Less Simple Example

In more complicated application areas, it becomes difficult or even impossible to capture all of the information about a state in a single term. This leads to a more sophisticated programming style in which one gives states arbitrary names and describes their properties in separate propositions.

As an example, consider the graph search problem described in the introduction. Instead of naming each state with a single term as in the 8-puzzle, each state is described using relations like the following.

`Marked(x,s)` means that `x` is marked in state `s`.
`Printed(x,s)` means that `x` has been printed in state `s`.

Just because two states agree on these relations does not mean they are identical. There may be other functions and relations on which they disagree. In fact, in general there is no guarantee of the existence of any *finite* set of functions and relations that distinguishes every state from every another, but in many applications a finite number of functions and relations is adequate to express the properties necessary to specify a procedure.

An action in this problem can be viewed as a transformation from a state in which some fact is false to a state in which that fact is true (or vice-versa) and in which as little else as possible has changed. In what follows, it is assumed that there is a relation called `Change` that holds between an action and any proposition that that action makes true.

This relationship between propositions and actions leads to a style of programming in terms of propositions rather than actions. In the axioms below, the relation Good is used to describe those propositions for which there is evidence of desirability, and the relation Bad is used to describe those propositions for which there is evidence of undesirablity. The following propositions define a procedure called F. The constraints prevent a machine from achieving any proposition that is known to be undesirable or that is not known to be desirable.

$$\texttt{Change}(\phi,\texttt{k}) \wedge \neg\texttt{Good}(\texttt{s},\phi) \Rightarrow \texttt{F}(\texttt{s})\neq\texttt{k}$$
$$\neg\texttt{Change}(\phi,\texttt{k}) \wedge \texttt{Good}(\texttt{s},\phi) \Rightarrow \texttt{F}(\texttt{s})\neq\texttt{k}$$
$$\texttt{Change}(\phi,\texttt{k}) \wedge \texttt{Bad}(\texttt{s},\phi) \Rightarrow \texttt{F}(\texttt{s})\neq\texttt{k}$$

It's important to keep in mind that the Good relation holds only of those propositions that ought to be *made* true in a state, and it does not necessarily hold of those propositions that ought to *be* true in that state. This relation is similar to McCarthy's Ought [McCarthy], except that he uses the word to refer directly to actions rather than to propositions.

Many programs have the property that there are no propositions (other than those above) that relate the facts true in one state to the facts true in another. This is the case, for example, of any program in which every proposition refers to state only in the form of a single, universally quantified state variable. In programs like this, it is possible to drop all references to state. Thus, statements like Marked(A,s) can be simplified to Marked(A); statements like Good(s,ϕ) can be simplified to Good(ϕ); and statements like Bad(s,ϕ) can be simplified to Bad(ϕ).

Using this simplified language, the following propositions show how a correct path finding program can be specified in PRO.

$$\texttt{Marked(A)}$$
$$\texttt{Marked(x)} \wedge \texttt{Conn(x,y)} \Rightarrow \texttt{Good(Marked(y))}$$
$$\texttt{Marked(J)} \Rightarrow \texttt{Good(Success)}$$

The first proposition simply states that the start node is marked. The second proposition states that if a node has an incoming connection from a marked node and is not known to be "disqualified" for any reason, then it ought to be marked. The final proposition expresses the program's ter-

mination condition. If the machine ever marks the goal node, then there is a path and the search is successful.

This program is sound in that it will succeed only if there is a path from A to J. However, it is not complete. In executing the program a machine might repeatedly mark a node it has already marked. As a result it may not find a path even if one exists and the set of nodes is finite. This problem can easily be fixed by augmenting the program with the following proposition. The effect is to disqualify any node that has already been marked. Remember that if a proposition is "bad", that doesn't mean it should be made false, only that there should be no effort to make it true.

```
Marked(y) ⇒ Bad(Marked(y))
```

With the addition of this proposition, the program is both correct and complete, assuming finiteness of the node set. However, the program is not completely specified in that there may be more than one node to be marked at any given time. In some cases this choice can be constrained as well. For example, the following proposition induces an ordering on node markings on the basis of the "quality" Q of each node. Other sorts of ordering information can be taken into account in similar fashion.

```
Q(x)>Q(y) ∧ Good(Marked(x)) ⇒ Bad(Marked(y))
```

Finally, a modest opportunism can be introduced into the program by extending the set of desirable state transformations. The first proposition below states that it is desirable to have a message printed if a node of quality zero is ever encountered. The second proposition ensures that this will happen before any further marking actions.

```
Marked(x) ∧ Q(x)=0 ∧ ¬Printed(x=0) ⇒ Good(Printed(x=0))
Good(Printed(x)) ⇒ Bad(Marked(y))
```

The main thing to note about this example is the incremental development of the program. The first few propositions define an initial set of "legal" actions without specifying any particular order in which to try them. The succeeding axioms constrain that order. Absolute prohibitions can be expressed by disqualifying nodes. At any point additional legal actions can be added, as with the printing actions at the end.

This example is especially simple in that the target machine is assumed to be capable of distinguishing those states in which two nodes are connected from those states in which they are not. As a result, one can use connectedness directly in specifying control rules. In many applications, this sort of information is not directly available and can only be obtained by executing appropriate sensory actions. For example, a Turing machine cannot directly distinguish the contents of any tape square except the one directly under its head. In general, in order for a property to be usable in prescribing control information, there must be a finite sequence of actions that transforms two states which differ on the property into two distinct partitions of the state space. For example, in any Turing machine there is a finite sequence of actions capable of bringing any tape square under the machine's head, thereby allowing the machine to distinguish the contents of that square.

The preceding program can be modified to include sensory actions, so that, even if a machine were not able to determine connectivity directly, it could still be used in prescribing control rules. The modification assumes the existence of a directly observable relation `Knownconn` and a sensory action called `Getconns`. `Knownconn` is empty in the initial state, i.e. no nodes are known to be connected. The effect of executing `Getconns(x)` in state `s` to produce state `t` is that `Knownconn(x,y)` is true for every `y` such that `Conn(x,y)`. `Getconns(x)` also marks node `x` as having been "seen", which means that the `Change` relation holds between a proposition of the form `Seen(x)` and the action `Getconns(x)`.

```
Change(Seen(x),Getconns(x))
```

Of course, `Knownconn` implies `Conn`. If in any state two nodes are known to be connected, then they are in fact connected. The following proposition expresses this fact and thus establishes the connection information necessary for the connection axiom at the beginning of this example to apply.

```
Knownconn(x,y) => Conn(x,y)
```

Given `Knownconn` and `Getconns` defined in this way, the following propositions permit the preceding program to find paths even for machines unable to discern connectivity directly. The first proposition ensures that no node is marked until it has been examined. The second proposition prescribes

that any frontier node be examined for connections and be recorded as having been examined.

```
¬Seen(x) ⇒ Bad(Marked(x))
Good(Marked(x)) ∧ ¬Seen(x) ⇒ Good(Seen(x))
```

This program is especially interesting because it illustrates how a finite state machine operates in an application area characterized by more states than there are in the machine. In this case the use of a sensory action allows the machine to "internalize" information about the external world long enough to govern its actions. Although the example doesn't illustrate this, after this information has been used it can be discarded to make room for subsequent sensory data.

5 Least Simple Example

The key characteristic of traditional programming languages is the wide variety of useful control primitives they provide, e.g. block statements, conditionals, and iteration statements. The advantage of having these primitives is ease in the specification of control information. The disadvantage is lack of flexibility where partial specifications of control are desirable. Fortunately, it is possible to define complete control primitives as partial programs and to integrate them with partial control primitives in a way that lends itself to a hybrid programming style.

As an example, consider the partial program in figure 7. The definition of NDPATHP is the same as the definition of PATHP in figure 3 except that the deterministic control primitive FORONE has been replaced by the nondeterministic primitive NDFORONE. Like FORONE, NDFORONE processes the elements of the list defined by its second argument to find one that satisfies the condition defined by its third argument. The difference is that FORONE does this processing left to right, by definition, whereas NDFORONE has no such constraint. Instead, the user constrains the order by writing separate propositions like the ones at the end of figure 7.

This section shows how complete control primitives like IF and FORONE can be defined as partial programs. That this can be done is not surprising in itself. What's interesting is the subsequent relaxation of these definitions

```
NDPATHP(X,Z) :=
   IF X=Z THEN RETURN(TRUE)
          ELSE NDFORONE(Y,CONNECTIONS(X),PATHP(Y,Z))

BEFORE(C,D)

Q(x)>Q(y) ⇒ BEFORE(x,y)
```

Figure 7: A partial path finding program

to allow the specification and integration of partial control primitives like NDFORONE.

In what follows, a traditional program is assumed to be a term that in any environment designates a specific sequence of actions for a machine to perform. For example, in an environment in which the variable A has value 2, the program fragment PRINT(A*3+A*2) designates the actions of multiplying 2 by 3, multiplying it 2 by 2, adding the results together, and printing the total.

This ordering of events in time is fundamental to traditional programming. In the most general formulation, time can be broken into a linearly ordered set of discrete points, and a function can be defined that associates a specific point in time with each state. The result of executing an action is a state that reflects the effects appropriate to that action and in which time has advanced to the next time point. The ordering of actions is enforced by constraining their execution with respect to this temporal ordering.

A formulation of this sort is fully adequate to define the control primitives found in traditional programming languages. However, the definitions are somewhat cumbersome, and for this reason a slightly different formulation is used in the definitions below.

In particular, the set of time points is replaced by a set of stack snapshots, and the time function is replaced by a relation between states and snapshots. The relation is necessary, since there may be more than one active stack. The result of executing an action is a state that reflects the results of that action and in which the appropriate stack has advanced one step.

Every stack snapshot is a sequence of stack frames, each of which corresponds to a subexpression of the program being executed. The topmost stack frame always designates a primitive machine operation. Every other stack frame is a sequence consisting of a type designator and an appropriate number of arguments.

Figure 8 shows stack snapshots and results at two points in the execution of the program `PRINT(A*3+A*2)`. The topmost stack frame in the first snapshot designates the action of multiplying 2 by 3, a primitive machine action. The next stack frame corresponds to the enclosing addition. The first item in the stack frame designates its type, in this case `SUBR` to differentiate it from the conditional and iteration stack frames defined below. The second item is the name of the operation to be performed. The third item is a list of as yet unprocessed arguments (here `A*2`), and the fourth is a list of values for the processed arguments (none so far). The final item is a list of variable bindings. The result box below the stack is empty, since no actions have been executed.

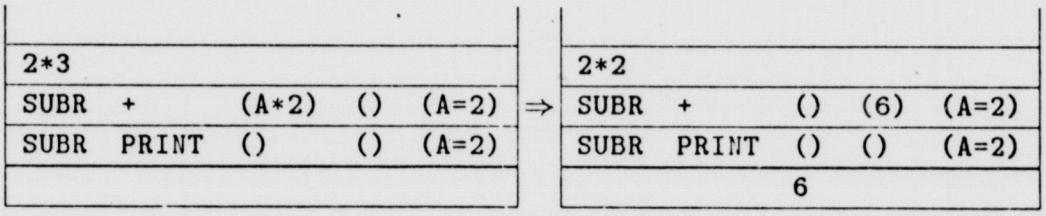

Figure 8: A stack snapshot

The progression of stack snapshots is straightforward. After each action is performed, the result is placed in the result box and the next action becomes the top stack frame in the resulting state. In this case, after the multiplication is done, the result is placed in the result box and in the list of processed arguments in the penultimate stack frame, and the next unprocessed argument becomes current. After the second multiplication is done, the addition becomes current. Once the addition is done, the printing action becomes current. After the printing action is done, the stack becomes empty.

A stack can be described in PRO as a list of stack frames, in which each stack frame is described by a single term. For example, the action in the leftmost frame above can be described by itself, i.e. 2*3. The next stack frame can be described by the term Subr(+,(A*2 1),(),(A=2)). The bottom frame can be described by the term Subr(PRINT,(),(),(A=2)). The executability of a stack named c in a state named s can be written Stack(c,s). The result z of the action leading to state t can be written Result(t)=z.

The following proposition is a partial definition of a procedure named F. It asserts that any action that is not the topmost frame of an active stack is illegal. If there is only one active stack in a state, the value of F is constrained to be the topmost action. If several stacks are active, the choice of action is left open. The "only if" $\Leftarrow$ operator is used here and in the subsequent propositions to simplify the presentation.

F(t)$\neq$k $\Leftarrow$ $\neg$Stack(t,(k.1))

A stack can become active in one of three ways. If a program x "doable" in state t and d is the stack corresponding to program x, then d is active in t. If an action k is executed in state s with stack (k.c) and has result z in state t and if d is the successor stack to (k.c) given result z, then d is active in state t. Finally, if an action k is executed in state s and c is a stack active in s with some action other than k on top, then c remains active in state t.

Stack(t,d) $\Leftarrow$ Doable(t,x) $\wedge$ Setup(x,(),(),d)
Stack(t,d)
  $\Leftarrow$ Do(k,s)=t $\wedge$ Stack(s,(k.c)) $\wedge$ Result(t)=z $\wedge$ Return(z,c,d)
Stack(t,c) $\Leftarrow$ Do(k,s)=t $\wedge$ Stack(s,(j.c)) $\wedge$ j$\neq$k

The relation Setup(x,c,v,d) asserts that d is a stack for the execution of program x in the context of stack c with variable binding environment v. The following propositions define Setup for atoms, numbers, quoted expressions, and primitive or defined operators.

Setup(x,c,v,d) $\Leftarrow$ Number(x) $\wedge$ Return(x,c,d)
Setup(x,c,v,d) $\Leftarrow$ Symbol(x) $\wedge$ Return(Assqval(x,v),c,d)

```
Setup('x,c,v,d) ⇐ Return(x,c,d)
Setup(g(x.l),c,v,d)
  ⇐ ¬Special(g) ∧ Setup(x,(Subr(g,l,(),V).c),v,d)
Assqval(p,(p.v).al)=v
Assqval(p,al)=v ⇒ Assqval(p,q.al)=v
```

The relation `Return(z,c,d)` asserts that `d` is a stack for the next step of stack fragment `c`. The first proposition below deals with the case of returning a value to a stack frame with unprocessed arguments. The value is added to the list of processed arguments, and the first unprocessed argument is set up. The second proposition asserts that, after all of the arguments of a primitive operator are processed, the action of applying that operator to its arguments becomes the topmost stack frame. The third proposition defines the handling of defined functions. Once all of the operator's arguments are processed, the variables of the operator's definition are bound to the values and the body of the definition is set up.

```
Return(z,(Subr(f,y.l,m,v).n),d)
  ⇐ Setup(y,(Subr(f,l,z.m,v).n),v,d)
Return(z,(Subr(f,(),M,v).n),(f(z.m).n))
  ⇐ Primitive(f)
Return(z,(Subr(f,(),M,v).c),d)
  ⇐ f(l):=y ∧ Pairs(l,m,v)=w ∧ Setup(y,c,w,d)
Pairs((),vl,al)=al
Pairs(pl,vl,al,bl) ⇒ Pairs(p.pl,v.vl,al,(p.v).bl)
```

Control primitives other than simple subroutine calling require specialized stack frames. One defines a new control primitive by describing the format for the associated stack frame, by describing how subexpressions involving the control primitive are set up, and by describing how answers are returned to it.

The `IF` primitive has a stack frame of the form `Ifblock(q,r,v)`, where `q` is the program to be executed if the condition is true, `r` is the program to be executed if the condition is false, and `v` is a variable binding list. The first proposition below asserts that an expression of the form `IF p THEN q ELSE r` is set up by adding an `Ifblock` stack frame to the stack and setting

up the predicate. If the value True is returned, the first branch is set up. If the value False is returned, the second branch is set up.

```
Setup(IF p THEN q ELSE r,c,v,d)
  ⇐ Setup(p,(Ifblock(q,r,v).c),v,d)
Return(True,(Ifblock(q,r,v).c),d) ⇐ Setup(q,c,v,d)
Return(False,(Ifblock(q,r,v).c),d) ⇐ Setup(r,c,v,d)
```

The FORONE primitive has a stack frame of the form Forblock(y,l,q,v), where y is the variable, l is the list of objects to be searched, q is the predicate to be applied, and v is a variable binding list. An expression of the form FORONE(y,p,q) is set up by establishing a Forblock stack frame and setting up the action to compute the list of objects. When a list of values is returned, the predicate is applied to the first element. If the result is False, the predicate is applied to the next element, and so on until the list is exhausted. If the predicate ever returns True, the successful result is passed up the stack.

```
Setup(FORONE(y,p,q),c,v,d)
  ⇐ Setup(p,(Forblock(y,(),Q,v).c),v,d)
Return(x.l,(Forblock(y,(),Q,v).c),d)
  ⇐ Setup(q,(Forblock(y,l,q,v).c),(y.x).v,d)
Return(False,(Forblock(y,l,q,v).c),d)
  ⇐ Setup(q,(Forblock(y,x.l,q,v).c),(y.x).v,d)
Return(True,(Forblock(y,(),Q,v).c),d) ⇐ Return(True,c,d)
Return(False,(Forblock(y,(),Q,v).c),d) ⇐ Return(False,c,d)
```

The handling of NDFORONE is similar to that of FORONE. The stack frame has the same arguments; the setup is similar; and success is handled in the same way. The main difference is that rather than processing the objects to be searched in order, a separate stack is set up for each element in the list.

```
Setup(NDFORONE(y,p,q),c,v,d)
  ⇐ Setup(p,(NDblock(y,(),Q,v).c),v,d)
Return(l,(NDblock(y,(),Q,v).c),d)
  ⇐ Member(x,l) ∧ Setup(q,(NDblock(y,m,q,v).c),(y.x).v,d)
Return(True,(NDblock(y,(),Q,v).c),d) ⇐ Return(True,c,d)
```

The multiple stacks set up in the handling of NDFORONE lead to multiple values for the procedure F defined above. The various choices for F can be ordered or pruned by writing additional constraints. For example, the following proposition connects the ordering of these choices to the ordering of objects described by the second and third propositions in figure 7.

```
F(t)≠NDPATHP(y,z) ⇐ Stack(t,(NDPATHP(x,z).c)) ∧ Before(x,y)
```

The key point of this example is to demonstrate that it is possible to define partial and complete control primitives in a way that allows one to mix the economical style of traditional programming with the flexibility of partial programming.

6 An Interpreter for Partial Programs

Even though a partial program is only a partial description of a procedure, it is possible to construct an interpreter that behaves in accordance with that description. Of course, in doing so the interpreter must occasionally make arbitrary choices. For example, in the absence of any ordering constraints, it may have to decide which of two possible actions to perform first. What's interesting about the interpreter is that if it is given a program augmented with a constraint that prohibits this arbitrary choice, it will adjust its activity accordingly.

The interpreter PRO takes as data a program name and a set of propositions. Its design assumes that there is a distinct term in the language for each partition and each action. The machine on which it is to run must be capable of observing the current partition and carrying out the prescribed action. And the machine must be capable of storing a program and drawing conclusions from it without affecting the state of the world.

A flowchart for PRO is given in figure 9. The interpreter is an exit-free loop. Each time around the loop the interpreter executes a single step of the program being run. The interpreter first observes the world to discover the current partition λ and records the relationship between the current state and this partition in the data base Γ. The interpreter then enumerates the machine's possible actions and for each such κ uses breadth-first resolution

in an attempt to prove $\pi(\sigma) \neq \kappa$. If this attempt succeeds, the action is inconsistent with Γ and is eliminated from consideration. If the inference procedure halts without proving this proposition, then κ must be consistent with Γ. Obviously, any of the inference attempts can run forever; and, in order to find a finite proof or a finite failure in finite time, the inferences must all be interleaved. If the a consistent κ is found, that fact is recorded in Γ and κ is executed. The interpreter then creates a new symbol for the resulting state, records the relationship between the old state and the new, and starts over again.

An *execution* of PRO is a partial function from states to actions. The function is defined in a state only if the machine running the program succeeds in finding an action to execute, in which case the value of the function is the action executed. If the inference procedure never terminates, the function is undefined. The start state of an execution is the state in which the execution began.

The PRO interpreter is sound in that, given any partial program, the actions it executes will be consistent with some procedure in the extension of that program.

Theorem 5 *If $\langle \pi, \Gamma \rangle$ is a partial program E is an execution of* PRO *on a program $\langle \pi, \Gamma \rangle$, then there is a procedure F in the extension of $\langle \pi, \Gamma \rangle$ that agrees with E wherever E is defined.*

Proof: Let E be an execution of PRO on the program $\langle \pi, \Gamma \rangle$ defined for successive states named $\sigma_1, \ldots, \sigma_n$ with corresponding actions $\kappa_1, \ldots, \kappa_n$. Let $\Gamma_0 = \Gamma$, and let $\Gamma_{i+1} = \Gamma_i \cup \{\pi(\sigma_i) = \kappa_i\}$. The only way for there not to be a subroutine consistent with E in the extension of this program is for at least one of the state-action pairs to be provably unacceptable, i.e. $\Gamma \models (\pi(\sigma_1) \neq \kappa_1 \vee \ldots \vee \pi(\sigma_n) \neq \kappa_n)$. Because of the deduction theorem and the above defintion of Γ_n, this is equivalent to $\Gamma_n \models \pi(\sigma_n) \neq \kappa_n$. But, if this were true, the resolution theorem prover would have discovered a proof to this effect and κ_n would not have been executed in state σ_n. □

Unfortunately, this result is slightly misleading. Since a partial program does not necessarily contain complete knowledge about the state of the world, it may impossible for it to prove the inconsistency of an action with the current state, and it may execute that action even though it is in fact

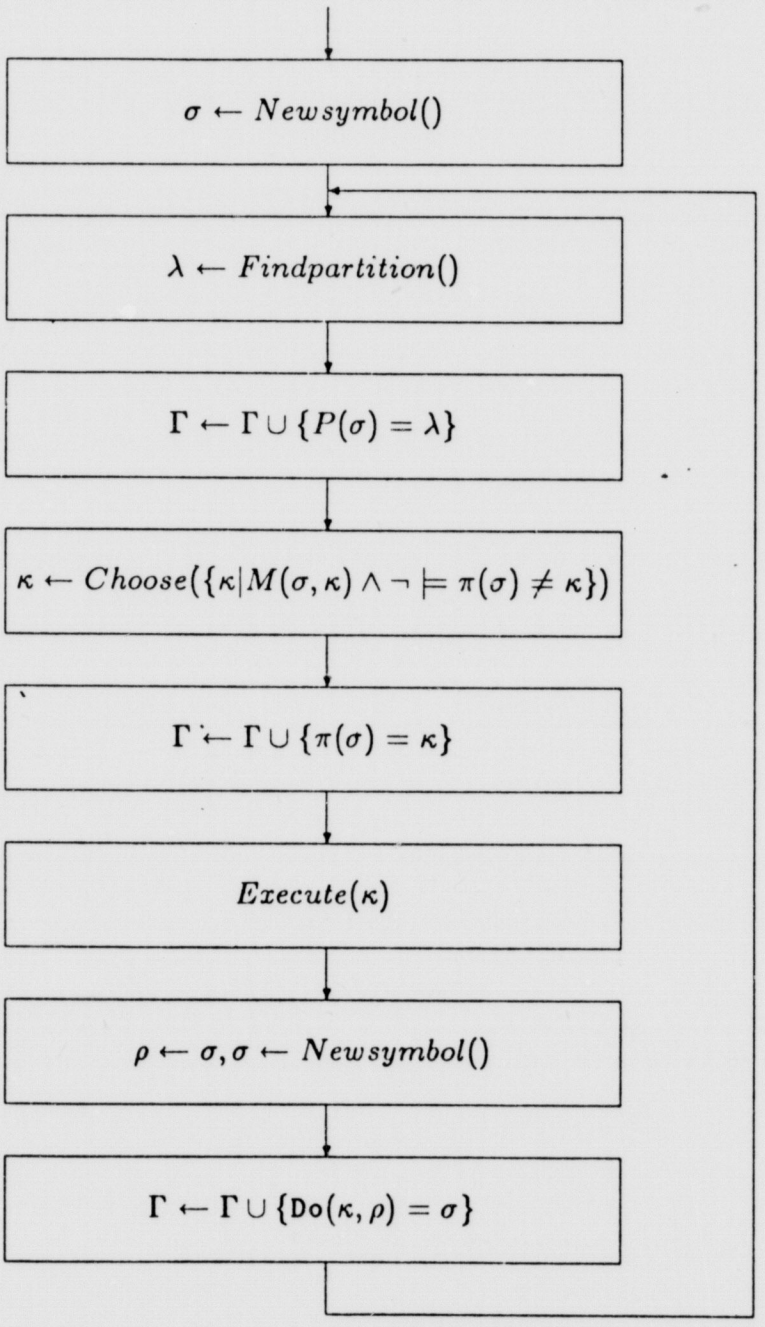

Figure 9: A flowchart for the PRO interpreter

incorrect. In the absence of the knowledge necessary to decide on an action, it makes an arbitrary choice, and this arbitrary choice may be wrong. For example, consider a program that is constrained not to invest in the stock market if President Reagan is sitting at his desk. If the program does not contain information about the President's current posture, it may decide to buy stock, even though the President is in fact sitting at his desk.

An arbitrary decision like this one can be catastrophic if at some later point the relevant information becomes available, say through the interpreter's observation step; for then the data base becomes inconsistent, the interpreter can prove the inconsistency of every action, and it ceases to act. A catastrophe like this one can be avoided by ensuring that all programs have the property described in section 4, i.e. the propositions comprising the program relate the choice of action in each state to the properties of that state alone. So long as a program has this property, subsequent observations cannot invalidate previous decisions, and no inconsistency results. Note this does not mean that the program will act correctly, only that the inconsistency will not be detected and the machine will not cease to operate and its operation will be consistent with the statements in the original program.

Another problem with the PRO interpreter is that it is not complete. There are programs with non-empty extensions on which the interpreter will loop indefinitely without executing any actions. The reason for this of the semidecidability of logical implication. The interpreter's inference procedure may loop forever in attempting to prove the consistency of an action in a state and thus may never act, even though the action is not provably inconsistent. Of course, there are well-known classes of propositions for which the consistency question is decidable. However, in general little more can be said than the following.

Theorem 6 *If $\langle \pi, \Gamma \rangle$ is a partial program and there is a procedure for determining in finite time that $\neg(\Gamma \models \pi(\sigma) \neq \kappa)$, then the interpreter will act in that situation.*

Proof: A consequence of the completeness of resolution and the interleaving of inference steps in PRO. □

The major disadvantage of the PRO interpreter for practical purposes is its inefficiency. Fortunately, it's possible to modify the interpreter to mitigate this problem if one is willing to sacrifice the soundness and completeness results or to restrict one's self to writing a more restricted class of programs.

A large part of the interpreter's inefficiency results from the breadth-first resolution procedure. One can save significant computation by replacing it with a depth-first backward-chaining procedure similar to that used in PROLOG. This procedure is powerful yet fairly easy to understand, and programmers can readily adapt to its computational constraints. In addition, there is significant computational advantage in caching selected results [Lenat, Hayes-Roth, Klahr].

Another source of inefficiency in PRO stems from the computation of a term to describe the current partition. In many programs only a portion of the information about the current partition is necessary to decide on an action to perform, yet PRO computes a term to describe all of th information about the partition. An alternative is to substitute procedural attachments for those functions and relations that the machine can distinguish. In this way the interpreter doesn't have to assemble a description of the entire partition in order to check a single property (e.g. CONN).

Note that the interpreter selects an action only after verifying that it isn't inconsistent with the axioms defining the program. In many cases, the axioms allow one to determine directly which action must be executed. This suggests the addition of another inference step to the interpreter just before the consistency tests intended to find a κ such that $\models \pi(\sigma) = \kappa$. If such an action is found, that action can be executed and the consistency tests can be bypassed.

Finally, it is acceptable to discard all information about the interpreter's execution history so long as one's programs have the property described in section 4. If the choice of action in every state is independent of the choice of action in every other state, then the historical record is unnecessary. All of the sample programs in this paper have this property.

7 Conclusions

The practicality of partial programming has been tested by its use in the construction of a number of different programs. These include an automated diagnostician for computer hardware faults (DART [Genesereth 1982]), a simulator for digital hardware (MARS [Singh]), a calculus program (MINIMA [Brown]), and an infectious disease diagnosis/tutoring system (NEOMYCIN [Bock, Clancey]). All of these programs were built with the help of MRS [Genesereth, Greiner, Smith], a declarative programming system with PRO as its control language. These experiments gave considerable insight into the advantages and disadvantages of partial programming and PRO in particular.

The primary advantage of partial programming is that it facilitates incremental system development. In the absence of a language in which partial specifications of behavior can be expressed, many arbitrary decisions must be made in implementing a program, thus inhibiting incremental program development. Using a partial programming language like PRO one can start with a search-intensive partial program and incrementally add constraints until one arrives at a (possibly search-free) traditional program. This ability to program "by subtraction" is especially important in building "knowledge-based" programs in which the knowledge to be built into a program is not fully known at the start of program development. Furthermore, partial programming allows one to use multiple programming styles, including state transition tables, demons, and traditional programming.

The key problem with any partial programming system is inefficiency in the interpreter. There is a substantial overhead in interpreting control axioms, and even for small applications this overhead is quite noticeable. On the other hand, the flexibility of the language facilitates and promotes the specification of control information that in many cases can lead to combinatoric savings that more than offsets this overhead (see [Smith and Genesereth], [Genesereth and Smith] for examples and a fuller discussion).

The most important direction for future work on partial programming is the development of a compiler able to transform partial programs into complete programs and implement them in traditional programming languages. In order to maintain the advantages of incremental program development

in the face of subsequent additions to a program, a partial programming system must be able to detect and eliminate inconsistencies between these additions and any arbitrary decisions made during compilation. This is quite simple so long as the compiler records dependency information and the system uses an appropriate truth maintenance routine.

In summary, the key idea in this paper is a view of programs as a set of constraints on the potential actions of a machine. A partial programming language like PRO allows one to write programs in which no arbitrary control commitments need be made. This fosters program integration and incremental system development. While there is a substantial overhead in interpreting partial programs, the flexibility facilitates the specification of control information that can in many cases more than offset this overhead.

Acknowledgements

The work reported in this paper is the end result of several years of research. The general idea of partial programs emerged from earlier work on the control of reasoning done jointly with Dave Smith. Russ Greiner and Dave Smith also contributed extensively to the research by helping in the design and implementation of the MRS system. Tom Dietterich, Jeff Finger, Matt Ginsberg, and Dave Smith read preliminary drafts of the paper and provided valuable feedback on the presentation. Support for the research was provided by the Office of Naval Research under contract number N00014-81-K-0004.

References

C. Bock, W. J. Clancey: "MRS/NEOMYCIN: Representing meta-control in predicate calculus", HPP-82-31, Stanford University Heuristic Programming Project, December 1982.

D. Brown: "MINIMA", Teknowledge Inc., 1982.

M. R. Genesereth: "The Role of Design Descriptions in Automated Diagnosis", to appear in *Artificial Intelligence*, 1985.

M. R. Genesereth, R. Greiner, D. E. Smith: "MRS - A Meta-Level Representation System", HPP-83-27, Stanford University Heuristic Programming Project, 1983.

M. R. Genesereth, D. E. Smith: "Procedural Hints in the Control of Reasoning", forthcoming publication.

D. B. Lenat, F. Hayes-Roth, P. Klahr: "Cognitive Economy", HPP-79-15, Stanford University Heuristic Programming Project, June 1979.

J. McCarthy: "Programs with Common Sense", Proceedings of the Teddington Conference on the Mechanization of Thought Processes, H. M. Stationery Office, London, 1960. Reprinted in Semantic Information Processing, M. Minsky (Ed), MIT Press, Cambridge, 1968, pp 403-410.

N. Nilsson: *Priciples of Artificial Intelligence*, Tioga Press, 1980.

N. Singh: "MARS: A Hierarchical Simulator for Digital Circuits", Stanford University Heuristic Programming Project., 1982.

D. E. Smith, M. R. Genesereth: "Ordering Conjuncts in Problem Solving: Serious Applications of Meta-Level Reasoning I.", to appear in *Artificial Intelligence*, 1985.