MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 464 May 1978

Comparative Schematology

Michael S. Patterson and Carl E. Hewitt

While we may have the intuitive idea of one programming language having greater power than another, or of some subset of a language being an adequate "core" for that language, we find when we try to formalize this notion that there is a serious theoretical difficulty. This lies in the fact that even quite rudimentary languages are nevertheless "universal" in the following sense. If the language allows us to program with simple arithmetic or list-processing functions, then any effective control structure can be simulated, traditionally by encoding a Turing machine computation in some way. In particular, a simple language with some basic arithmetic can express programs for any partial recursive function. Such an encoding is usually quite unnatural and impossibly inefficient. Thus in order to carry on a practical study of the comparative power of different languages we are led to banish explicit functions and deal instead with abstract, uninterpreted programs, or schemas. What follows is a brief report on some preliminary exploration in this area.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-75-C-0522.

Comparative Schematology

Michael S. Paterson and Carl E. Hewitt
Project MAC, M.I.T., Cambridge, Massachusetts

Introduction

While we may have the intuitive idea of one programming language having greater power than another, or of some subset of a language being an adequate 'core' for that language, we find when we try to formalize this notion that there is a serious theoretical difficulty. This lies in the fact that even quite rudimentary languages are nevertheless 'universal' in the following sense. If the language allows us to program with simple arithmetic or list-processing functions then any effective control structure can be simulated, traditionally by encoding a Turing machine computation in some way. In particular, a simple language with some basic arithmetic can express programs for any partial recursive function. Such an encoding is usually quite unnatural and impossibly inefficient. Thus in order to carry on a practical study of the comparative power of different languages we are led to banish explicit functions and deal instead with abstract, uninterpreted programs, or schemas. What follows is a brief report on some preliminary exploration in this area.

Languages

The simplest language we shall study is a flow-chart language with which we write program schemas such as shown in Figure 1. When we have provided an interpretation for

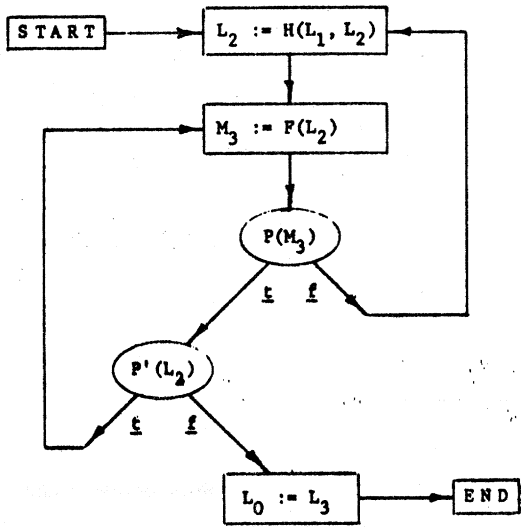

Figure 1. A program schema.

the basic function symbols and predicate symbols involved, a program schema may be regarded as an executable program defining a partial function. As a convenient convention we take the arguments of this function to be the initial values of locations L_1, L_2, $\cdots$ (up to as many as are mentioned) and the value as being the final value of L_0 if this is defined. Other locations M_0, M_1, $\cdots$, N_0, N_1, $\cdots$ are used only as 'working space'. We may as well assume that none of these latter locations is used as an argument before being assigned to.

Another language we shall use provides for the recursive definition of functions using conditional expressions. For example:

$$f(x_1, x_2) = \text{if } g(x_2) \text{ then } F(\text{if } P(x_1) \text{ then } x_2 \text{ else } A) \text{ else if } P(x_2) \text{ then } \underline{\text{true}} \text{ else } f(x_2, F(x_1))$$

$$g(x_1) = \text{if } P(f(x_1, F(x_1)) \text{ then } \underline{\text{false}} \text{ else } g(A)$$

defines two abstract functions by simultaneous recursion. Given an interpretation of the basic function and predicate symbols (which are the upper-case symbols in the definitions), such a system defines a partial function corresponding to each equation. The system is deemed to compute the function given by the first definition. Such a system is called a <u>recursive schema</u>. If there is only one equation it is a <u>simple</u> recursive schema, if several it is <u>compound</u>.

Interpretations and Equivalence

An <u>interpretation</u>, I, for program schemas and recursive schemas provides:

(i) a domain D

(ii) for each basic predicate symbol P, a (total) predicate

$$P_I : D \to \{\underline{\text{true}}, \underline{\text{false}}\}$$

(iii) for each n-ary basic function symbol ($n \geq 0$) F, a (total function)

$$F_I : D^n \to D$$

In general we use P, P', $\cdots$ as predicate symbols and other capital letters as function symbols. A constant is a 0-ary function.

The partial function defined by schema S under interpretation I is denoted by S_I. For two partial functions u, v, we write $u \simeq v$ if, for all x, either both of u(x), v(x) are undefined or both are defined and u(x) = v(x). Two schemas S, S' (not necessarily of the same type) are (strongly) <u>equivalent</u>, $S \equiv S'$ if, for all interpretations I, $S_I \simeq S'_I$.

Let $\mathcal{S}$, $\mathcal{S}'$ be two classes of schemas. We write $\mathcal{S} \preccurlyeq \mathcal{S}'$ if $(\forall S \in \mathcal{S})(\exists S' \in \mathcal{S}')[S \equiv S']$ and $\mathcal{S} \prec \mathcal{S}'$ if $\mathcal{S} \preccurlyeq \mathcal{S}'$ but not $\mathcal{S}' \preccurlyeq \mathcal{S}$. Let $\mathcal{R}$ be the class of recursive schemas and $\mathcal{P}$ be the class of program schemas.

Theorem 1. $\mathcal{P} < \mathcal{R}$.

Proof. To show $\mathcal{P} \prec \mathcal{R}$ involves only a routine construction which we outline below. (For further details see [1].) For each flowchart box b_i in program schema U, consider the abstract partial function f_i, which for given values of all the locations is computed as follows. Start the schema at b_i with the locations having these values, and then the value of the function is the final value of L_0, if defined. It is easy to write a compound recursive schema which defines the f_i recursively. If b_0 is the START box then the function corresponding to U is got from f_0 by simply suppressing some of its arguments.

Now to show that the containment is proper, consider the following recursive schema V.

$$\underline{V}. \qquad f(x) = \text{if } P(x) \text{ then } x$$
$$\text{else } H(f(L(x)), f(R(x)))$$

We shall show that no program schema can be equivalent to V.

A useful notion is that of a free interpretation. In a free interpretation the domain is the set of all strings composed from the basic function symbols. Then, for example, if H is a binary function symbol and E_1, E_2 are strings,

$$H_I(E_1, E_2) = H\,E_1\,E_2 \text{ for any free interpretation I.}$$

The interpretation of the predicate symbols is unconstrained. It should be clear that for most purposes, it is sufficient to consider only free interpretations. For example, for any S, S', $S \equiv S'$ if and only if $S_I = S'_I$ for all free I.

For the present proof (and the next one) we consider the family of free interpretations $\{I_n\}_{n>0}$ where

$$P_{I_n}(E) = \underline{\text{true}} \text{ if length}(E) = n$$
$$= \underline{\text{false}} \text{ otherwise.}$$

What is the value of $V_{I_n}(\lambda)$ where λ is the empty string?

For n = 1, the value is HLR,
for n = 2, it is HHLLRLHLRRR
etc.

We show that to compute the value for I_n at least n + 1 locations are required for working space. It helps to present the same situation in a geometric form. We are to play a game of placing movable tokens on a finite binary tree, T_n. For n = 3, the tree T_3 is given in Figure 2. The rules of the game are that any token may be put on a bottom node at any time. If the two nodes below a given node are covered then any token may be put at the given node. How many tokens are needed to be able to reach the top node? It is easy to see that n + 1 are sufficient in general for T_n, and the following

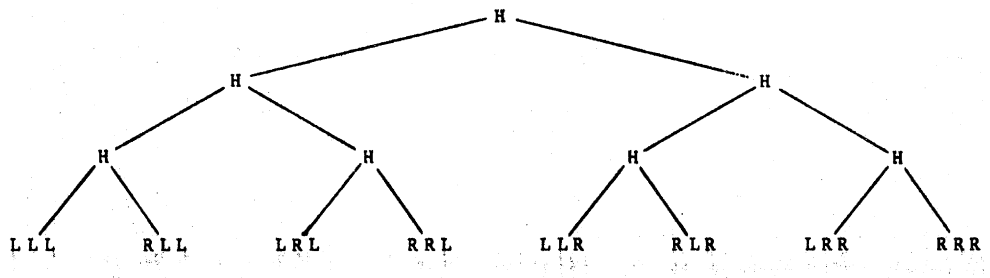

Figure 2. Binary tree for n = 3.

argument establishes their necessity.

A tree will be said to be closed at a given stage of the game if there is at least one token on each path from top to bottom. Initially, the tree is devoid of tokens and so is not closed. Finally there is a token at the top node and so it is closed. We concentrate our attention on the time at which the tree first becomes closed. This can only happen as a result of placing a token at a bottom node, closing off the last path. Now this path is otherwise empty of tokens and so each of the n sub-trees sprouting immediately off this path must be independently closed. Since at least one token is needed to close any tree, there are at least n+1 tokens on the tree at this time.

The relation between a computation of a program schema and moves in this game we assume to be self-evident. Suppose a program scheme U is equivalent to V and has just r locations. This is immediately absurd because there is no way for U to compute the required output for the interpretation I_r. This completes the first proof.

Program schemas are therefore unequal to the task of computing certain abstract functions for the simple reason that a fixed number of locations cannot compute all the necessary final values. This is not a very interesting reason so we show in a second proof of the same theorem that program schemas can fail for more subtle reasons which reflect the inadequacy of their control structure. The "target" schema for this proof computes a partial predicate and is got from V, more or less, by replacing H by the Boolean function and.

Second Proof of Theorem 1. Consider the recursive schema:

W. f(x) = if P(x) then true
else if f(L(x)) then f(R(x)) else false

Since the value of this schema is either true or is undefined, no argument of the kind given above is useful. Suppose program schema U is equivalent to W and has t boxes and r locations, and we may suppose, without loss of generality, that the only predicate and function symbols occurring in U are P, L, R. A state of U under a given interpretation is specified by a box of U together with a value from the domain for each location. Two states, S_1, S_2 are (just-for-now) equivalent if the sequences of boxes for the

computations continuing from S_1, S_2 are the same. Consider the equivalence classes of states of U under the interpretation I_n defined above. With a little thought one can see that the only property of the value of a location which can affect the equivalence class is the length of the string. Furthermore any two strings of length greater than n are indistinguishable by P either immediately or in their future. Thus U has at most $t\cdot(n+2)^r$ equivalence classes under I_n. If, during a computation of U an equivalence class is repeated, then the computation is doomed to loop. $U_{I_n}(\wedge)$ is supposed to terminate (with value true), hence must never repeat an equivalence class, and therefore runs for no more than $t\cdot(n+2)^r$ steps. If n has been chosen sufficiently large that

$$2^n > t\cdot(n+2)^r$$

then U cannot have time to test each one of

$$P(\overleftarrow{L\,L\,\cdots\,L}^{n}),\ P(\overleftarrow{R\,L\cdots\,L}^{n}),\ \ldots\ldots,\ P(\overleftarrow{R\,R\cdots\,R}^{n})$$

but nevertheless halts with value true. If we slightly modify I_n by letting P be false for some such expression which is not tested, then of course U will never notice and still give the value true, but the value of W is now undefined. Thus $U \not\equiv W$ and the proof is complete. (Of course this method of proof works equally well for schema V.)

Linearly Recursive Schemas

The task of effectively characterizing those recursive schemas which are "programmable" is impossible. Indeed, if $\mathcal{P}^* \subseteq \mathcal{R}$ is the subset of those recursive schemas which are equivalent to some program schema, we can neither effectively enumerate $\mathcal{P}^*$ nor $\mathcal{R} - \mathcal{P}^*$. This follows from the application of straightforward techniques which are described in [1] or [2] and will not be proved here. The best that we can hope for then is to effectively characterize large sub-classes of $\mathcal{P}^*$ and $\mathcal{R} - \mathcal{P}^*$. At present we have a fairly extensive effective sub-class of $\mathcal{R} - \mathcal{P}^*$ which encompasses both V and W and which we are seeking to extend further.

On the other hand, an approximation to $\mathcal{P}^*$ is provided by the "linear" schemas which are described shortly. A recursive schema may be regarded as giving a way of computing the value of a non-basic function given the values of the same or other functions at other arguments. The determination of these latter values will in general require further recursive calls, and so on. For example, in the following schema:

$$f(x, y) = \text{if } P(x) \text{ then } f(x, S(y)) \text{ else}$$
$$\text{if } P(g(R(x))) \text{ then } S(x) \text{ else } H(g(R(x)), R(y))$$

$$g(x) = \text{if } P'(x) \text{ then } H(g(S(x)), R(g(S(x))))$$
$$\text{else if } P(x) \text{ then } f(x, x) \text{ else } A$$

the evaluation of f(x, y) might need the value of f(x, S(y)) or the value of g(R(x)), and the evaluation of g(x) might need the value of g(S(x)) or f(x, x). But notice in this example, and this is the defining criterion for a linear schema, that in any evaluation at most one further value of a non-basic function is immediately required.

Theorem 2. If X is linear then $X \in \rho^*$.

Rather than give a proof of this theorem, we shall give here an example of the translation into program schema form of a simple linear program. The proof of Theorem 2 however, is by reducing the general linear case to a schema isomorphic to this example. So the following translation or compilation may be regarded as the "canonical" example!

L^*. $\quad f(x) = \text{if } P(x) \text{ then } R(x) \text{ else } S(f(T(x)), x)$

The second argument of S is an important feature, adding to the difficulty of this example. Let $v_0, v_1, \cdots v_m$ be the values of the successive instances of f called for in some terminating evaluation. We observe that:

$$v_m = R_I(T_I^{(m)}(x))$$

$$v_r = S_I(v_{r+1}, T_I^{(r)}(x)) \qquad \text{for } r = 0, \cdots, m-1,$$

and further, that if only we had some form of counter able to count up to integers less than or equal to m, then we could easily evaluate f by computing in turn $v_m, v_{m-1}, \cdots, v_0$. However, such a counter can be simulated when we realize that:

$$P_I(T_I^{(r)}(x)) = \underline{\text{false}} \qquad \text{for } r = 0, \cdots, m-1$$

$$= \underline{\text{true}} \qquad \text{for } r = m.$$

An equivalent program schema to L^* is given in Figure 3.

Discussion

Of course the efficiency of the above translation leaves something to be desired, and we are paying for our restriction to only a finite number of locations by an increase in the computation time. The cause of the problem in this example is that we are required to compute, in order, the sequence:

$$T^{(m)}(x), T^{(m-1)}(x), \cdots, TT(x), T(x)$$

which the schema L does by computing each term independently all the way up from x, and thus requires of the order of m^2 operations. To investigate this situation of trading off between time and space in more detail we have considered the following simple combinatorial problem.

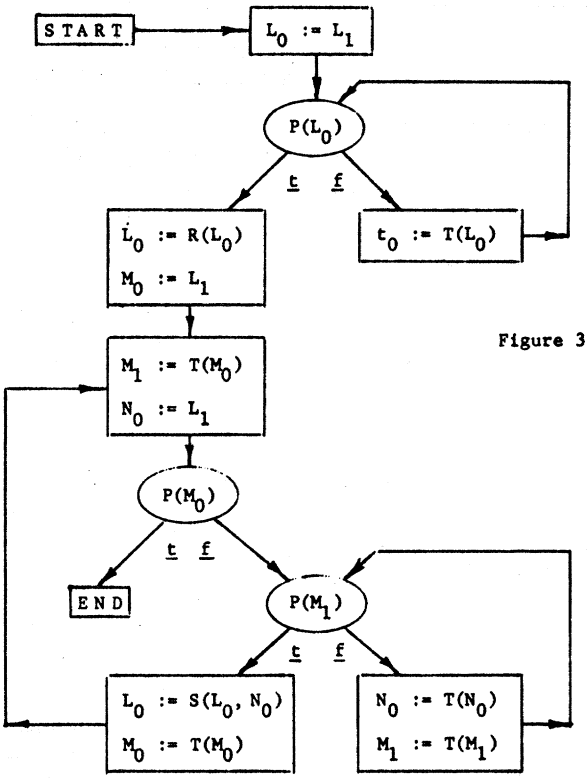

Figure 3. Program schema equivalent to a linear schema.

Suppose we have always available the value X in a "read-only" location, M_0, and we have in addition a further k locations, $M_1, \cdots, M_k$. For given fixed m, we want to write a shortest possible sequence of instructions of the form:

$$M_i := T(M_j) \qquad 0 < i \leq k,\ 0 \leq j \leq k$$

to compute successively the values

$$T^{(m)}(X), \cdots T\,T(X),\ T(X).$$

The values may appear in any location, but must be produced in the given order. Let L(k, m) be the length of the shortest such sequence of instructions. For example, $L(1, m) = \frac{1}{2}m\ (m+1)$. We have derived an explicit formula for L(k, m) from which we can show

$$L(k, m) \sim m^{1+1/k}$$

Thus there exists the possibility of computing linear recursive schematic functions using a fixed number of locations with reasonable efficiency. However, it is likely that some more versatile control structure than that of program schemas is required to realize this possibility.

We are currently investigating the relations between various augmented forms of program schema, such as program schemas with counters, push-down stores, or stacks. These augmentations may either be expressed by more complicated definitions of 'schema' or else may be got from the ordinary program schema by fixing part of the interpretation. For example, to express the idea of counters we could fix the interpretation of the constant symbol ZERO, function symbols ADD1, SUB1, and predicate symbol POSITIVE? in the obvious way. We must take care in applying some of the well-known results of automata theory to this area. For example, program schemas with counters are not 'universal' in this theory, because the first proof of Theorem 1 shows that no such schema can be equivalent to the recursive schema V. However, we can show that under an appropriate definition of push-down store program schemas, they are equivalent to the class of recursive schemas.

A Class of Parallel Schemas

For a very simple extension of recursive schemas we allow a parallel form of conditional expression.

IF p THEN g ELSE r

has the value g if p is true, the value r if p is false and in addition if p is undefined but g and r are both defined and are equal then the value is g. As a special case of this connective we define:

p /OR/ g to mean IF p THEN true ELSE g

So if either p or g has the value true then (p/OR/g) is true, and if both p and g are undefined it is also undefined. Let $\mathcal{S}$ be the class of schemas we get by extending $\mathcal{R}$ with /OR/.

Theorem 3. $\mathcal{R} < \mathcal{S}$.

The proof of this theorem which is too long and detailed to give here is by showing that no recursive schema can be equivalent to the following schema.

S. f(x) = if P(x) then true else (f(L(x))/OR/f(R(x)))

The value of $S_I(\Lambda)$ for a free interpretation I is true if there is at least one string of L's and R's for which P is true and is undefined otherwise. In the proof we characterize the behavior of recursive schemas under the free interpretation where P is always false. The only strings on the binary tree of (L, R)-strings that a recursive schema can "look at" are those strings within a bounded distance of a finite number of paths descending through the tree. Therefore, any recursive schema must fail to test certain strings and so will give a different value from S for some interpretations.

Conclusion

We have a clear notion of effective schemes of computation involving uninterpreted functions, and should like to have a fairly natural augmentation of program schemas capable of representing any such effective computation. A good candidate for this supreme position in the heirarchy of schemas would seem to be program schemas with two push-down stores. Provided the schema has the ability to put special control constants in its stores and to subsequently recognize them, the "universality" of this model appears assured.

Throughout this paper we have made the simplifying assumption that only total functions and predicates can appear in interpretations. Removing this assumption changes several of our results, and introduces new considerations. For example, to show that $\mathcal{R} < \mathcal{S}$ we have only to notice that the schema

$$f(x) = (P(x) \text{ /OR/ } P'(x))$$

has no recursive equivalent. Also the concept of effective computation is now no longer unambiguous and depends on the conventions we adopt concerning the evaluation of partial basic functions.

Acknowledgement

We should like to mention the work of Dr. H. Ray Strong [3] which discusses in more detail many of the topics of this paper, and to acknowledge the helpful discussions we have had with him.

References

1. D. C. Luckham, D. M. R. Park and M. S. Paterson, On formalized computer programs. To appear in the Journal of Computer and Systems Sciences, June 1970.

2. M. S. Paterson, Equivalence Problems in a Model of Computation. Ph.D. Thesis, Cambridge University, 1967.

3. H. R. Strong, Translating Recursion Equations into Flow Charts. Parts 1 and 2. Reports RC 2834 (March 1970) and RC 2859 (April 1970), IBM Research Center, Yorktown Heights, New York.