MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence
Memo No. 207 August 1970

MORE COMPARATIVE SCHEMATOLOGY

Carl Hewitt

Schemas are programs in which some of the function symbols are uninterpreted. In this paper we compare classes of schemas in which various kinds of constraints are imposed on some of the function symbols. Among the classes of schemas compared are program, recursive, hierarchical, and parallel.

***

Work reported herein was supported by the Artificial Intelligence Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002.

Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

Contents

0. Contents

1. Analytic Theory
    1.1 Classes of Schemata
        1.1.1 Recursive Schemas
            1.1.1.1 Comparison with Program Schemas
                1.1.1.1.1 Bushy Theorem
                1.1.1.1.2 Single Instance Theorem
            1.1.1.2 Compilation
            1.1.1.3 Schemas with Resets
            1.1.1.4 Decompilation
        1.1.2 Schemas with Counters
        1.1.3 Parallel Schemas
        1.1.4 Locative Schemas
        1.1.5 Schemas with Selectors and Replacement
        1.1.6 Schemas with Free Variables
        1.1.7 Hierarchical Schemas
            1.1.7.1 Comparison with Recursive Schemas
            1.1.7.2 Comparison with Parallel Schemas
            1.1.7.3 PLANNER Schemas
    1.2 Intentions
        1.2.1 Definition of Intentions
        1.2.2 Completeness of Intentional Analysis

2. Synthetic Theory
    2.1 Realizations
        2.1.1 Realizations for the Quantificational Calculus
        2.1.2 Realizations of PLANNER Theorems
    2.2 Construction of Schemas
        2.2.1 Completeness of Procedural Abstraction
        2.2.2 Completeness of Method of Canned Loops

3. Current Problems and Future Work

1. Analytic Theory

1.1 Classes of Schemata

1.1.1 Recursive Schemas

The following is an informal progress report of some work that I have done with Mike Paterson. John L. White made important suggestions and corrections. The result that recursive schemas are more powerful than program schema was obtained as a term project in the spring of 1969. Rigorous proofs are not given here but just an indication of how a proof would go. Program schemas are nonrecursive procedures that have uninterpreted function symbols and predicate symbols. We shall use capital letters to denote uninterpreted symbols. We shall allow schemas to use a finite number of distinguished objects which can be tested by the binary predicate "is". For example (is x "hello") is true only if x is the distinguished constant "hello". Functions evaluate their arguments from left to right.

The following is an example of a program schema:

```
(g x) = begin (register y)
        (comment y is a a register of the program schema g)
again:
        (if (or (P x) (is x "dolly")) then (return y))
        (x <- (L y))
        (y <- (R (R y)))
        (go again)
        end
```

A recursive schema is a program schema that is allowed to call itself or other recursive schemas recursively. The following is an example of a recursive schema k which is defined by a set of recursive equations:

```
(k x) = (if (P x) then x
                else (C (k x) (m (R x)))

(m y) = (if (P (R y)) then (L y)
                else (C (m (l y)) (k (k x))))
```

For any recursive schema defined by a set of recursive equations we can construct an equivalent recursive schema with only one equation and one additional argument to tell which equation is being simulated. This is possible because we allow recursive schemata to use a finite number of distinguished constants and predicates to test for these constants. The following is an example of a recursive schema that uses the interpreted constant symbols true and false.

```
(f x) = (if (P x)
             then
                (if (Q x)
                     then true
                     else false)
        elseif (f (L x))
                     then true
        else (f (R x)))
```

1.1.1.1 Comparison with Program Schemas

In fact the above recursive schema is not equivalent to any program schema. By equivalent we mean that the two schemas must both fail to terminate or both must return the same value for all interpretations of the functions P, Q, L, and R. Often we will take the set of uninterpreted terms as our domain of interpretation. In the above case the domain of interpretation is x, (L x), (R x), (L (L x)), (L (R x)), (R (L x), etc. The function letters L and R are interpreted as l and r where:

(l y) is defined to be the term (L y)
(r y) is the term (R y)

Thus (l (R (L x))) is the term (L (R (L x))). Two schemas are equivalent if and only if they define the same function on the domain of terms.

Theorem:

The function f defined above is not equivalent to any program schema.

Proof: Consider the following class of interpretations (I n) where n is a non-negative integer:

The domain of interpretation is the set of terms that can be constructed from the indeterminate x and the predicate letters L and R. The predicate Q is interpreted as a function q with range (true false). The predicate P is interpreted as the function p:

```
        (p (h!0 ...(h!m x)...)) = true for m = n
                                = false otherwise
```

where each h!i ("h subscripted by i") is the interpretation for R or the interpretation for L and there is at most one path such that

```
        (q (h!0...(h!n x)...)) = true
```

The domain of (I n) is the set of all terms that can be constructed from the indeterminate x and the functions L and R. We are going to prove that for any program schema P we can find an integer t such that P does not define not the same function as the recursive schema f on at least one member of the class (I t). In the the interpretation (I 3), we have the following L-R tree (where each node is a term in the domain of (i 3)):

```
(x      ((L x)
                ((L (L x))
                        ((L (L (L x))))
                        ((R (L (L x)))))
                ((R (L x))
                        ((L (R (L x))))
                        ((R (R (L x))))))
        ((R x)
                ((L (R x))
                        ((L (L (R x))))
                        ((R (L (R x)))))
                ((R (R x))
                        ((L (R (R x))))
                        ((R (R (R x)))))))
```

The function p is true only on the right-most (i.e. bottom) nodes and q is true on at most one of the right-most (bottom) nodes. We shall define the state of a program schema P at a point in its computation to be the contents of the registers of

P together with the statement of P that will be executed next. Two states S1 and S2 of P under the interpretion I will be said to be EQUIVALENT if p executes exactly the same sequence of instructions when started from S1 as when started from S2. Suppose we have a program schema P with s statements and k registers. In the interpretation (I n), the program schema P has at most s*((n+2)**k) equivalence classes of states where ** is the exponential function. (Effectively the only thing the schema can do is to count down each of its k registers to the bottom of the L-R tree and test each of them to see if it has reached the bottom.) However, a program schema needs at least 2**n steps in order to check if q is true on each of the nodes at level n. Therefore for sufficiently large n, P must go into an infinite loop since it will arrive at two distinct nodes in the same state. To see the matter somewhat differently look at the sequence of equivalence classes of states. If the sequence repeats then the program schema is in an infinite loop. But the poor program schema must seek and test all 2**n terminal nodes and then halt. Therefore the program schema needs at least 2**n equivalence classes.

We define the protocol tree of a recursive schema S to be the possibly infinite tree obtained by substituting for each schema that occurs in S the definition of that schema to obtain a schema S′. The whole process is then repeated for the schema S′. The protocol tree of the schema f is

```
(if (P x)
     then
        (if (Q x)
             then true
             else false)
     else
        (if (P (L x))
             then
                (if (Q (L x))
                     then true
                     else
                        (if (P (R x))
                             then
                                (if (Q (R x))
                                     then true
                                     else false)
                             else false))
             else
                (if (P (L (L x)))
                     then
                        (if (Q (L (L x)))
                             then true
                             else
                                (if (P (L (R x)))
                                     then
                                        (if (Q (L (R x)))
                                             then true
                                             else
                                                ...)
                                     else
                                        (if (P (R x))
                                                ...))
                     else
                        (if (P (R (L x))
                             then
                                (if (Q (R (L x)))
                                     then
                                        true
                                     else
                                        ...)
                             else
                                (if (P (R x))
                                     then
                                        (if (Q (R x))
                                             then
                                                true
                                             else
                                                false)
                                     else
                                        false)))
```

We shall say that a protocol is BUSHY if it contains an infinite binary subtree such below every node on the subtree there is a path which can converge. Notice that if a protocol tree is bushy then it is infinite.

The schema f defined above is bushy as is shown by the following bushy sub-tree.

```
((P x)
        ((P (L x))
                ((P (L (L x)))
                        (...)
                        ((...))
                ((P (R (L x)))
                        (...)
                        ((...)))
        ((P (R x))
                ((P (L (R x)))
                        (...)
                        ((...))
                ((P (R (R x)))
                        (...)
                        ((...))))
```

The schema f1 defined below is not b bushy and there is an equivalent program schema.

```
(f1 x)=(if (P x)
             then true
        elseif (f1 (L x)
             then true
        elseif (f1 (R x))
             then (f1 x)
             else false)
```

The protocol tree is not bushy since it does not contain an infinite binary tree.

```
if (P x)
     then true
     else
        if (P (L x))
             then true
             else
                if (P (L (L x)))
                     then true
                     else
                        ...
```

The following schema does not have a bushy protocol tree because none of the nodes have convergent paths below them.

```
(f2 x)=if (P x)
     then (C (f2 (R x)) (f2 (L x))
     else (C (f2 (L x)) (f2 (R x))
```

The schema ff defined below has a bushy protocol tree. Note the use of the uninterpreted function symbol "C".

```
(ff x) =
        (if (P x)
             then x
             else
                (C
                        (ff (L x))
                        (ff (R x))))
```

The protocol tree for the schema ff is:

```
(if (P x)
     then x
     else
        (if (P (L x))
             then
                (if (P (R x))
                     then (C (L x) (R x))
                     else
                        (if (P (L (R x)))
                             then
                                (if (P (L (L x)))
                                     then (C (L x) (C (L (r x))
(R (R x))))
                                     else
                                        ...)
                             else
                                ...))
             else
                (if (P (L (L x)))
                     then
                        (if (P (R (L x)))
                             then
                                (if (P (R x))
                                     then (C (C (L (L x)) (R x))
(R x))
                             else
                                ...)
                     else
                        ...))
```

The schema ff1 defined below does not have a bushy protocol tree.

```
(ff1 x)=(if
             (or (P x) (not (P (L x)))
             then x
             else
                (C (ff1 (L x)) (ff1 (R x))))
```

since it is equivalent to

```
(ff x) = if (P x) then x
        else (C (L x) (ff1 (R x)))
```

All of the above schemas given above have the property they if they are not bushy then they are not equivalent to any program schema.

```
(f4 x y) = if (P y) then x
             else
                (C
                        (f4 (L x) (G y))
                        (f4 (R x) (G y)))

(f5 x y) = if (P y)
             then
```

```
                if (Q y) then true else false
             else
                (and
                        (f5 (L x) (G y))
                        (f5 (R x) (G y)))
```

neither of the above schemas is equivalent to any program schema. The reader might consider how to extend the definition of bushy in order to encompass the above cases. [Hint: Consider protocol trees which have an infinite branch on which arbitrarily large "bushy" brances appear.]

1.1.1.1.1 The Bushy Theorem:

If a schema has a bushy protocol tree then it is not equivalent to any program schema.

The following is an example of a recursive schema which does not have a bushy protocol tree but is not equivalent to any program schema.

```
(f x y) = if (P y)
             then x
             else
                (C
                        (f (L x) (H y))
                        (f (R x) (H y)))
```

However, if we restrict ourselves to recursive schemata which have only one instance of a call to themselves then we can find an equivalent program schema. The program schema is obtained from the recursive schema by doing the tests in the opposite order. A more primitive form of the transformation is well known to compiler writers. The transformation explained below will work only in the absence of side effects. The running time of the transformed program is of the order of the square of the running time of the original program. We conjecture that there is no transformation that can do better than the square of the running time in general. For example consider

```
(f x) = (if (P x)
            then x
            else (H (f Kdelta x)) x))
```

1.1.1.1.2 The Single Instance Theorem:

A single recursive schematic equation that defines a function form f can be transformed into an equivalent program schema if the form f appears only once in the definition of the function.

Proof:

Define (F^n x) to be F applied n times to some argument x.

(F^0 x) = x

(F^(n+1) x) = (F (F^n x))

For example (F^1 x) is (F x) and (F^2 x) is (F (F x)).

Suppose the definition of f is of the form

```
(f k) =  if (alpha k)
             then (beta k)
             else (gamma (f (delta  k)) k)
```

where (alpha k) is the expression that is evaluated before the recursive call to f, (beta k) is the expression that is evaluated if there is no recursive call to f, and (gamma (f (delta k)) k) is the value for a recursive call to f. The protocol tree for f is

```
(if (alpha (delta^0 k))
     then
        (beta (delta^0 k))
     else
        (gamma
                (if (alpha (delta^1 k))
                     then
                        (beta (delta^1 k))
                     else
                        (gamma
                                (if (alpha (delta^2 k))
                                     then
                                        (beta (delta^2 k))
                                     else
                                        (gamma
                                                (if (alpha
(delta^3 k))
                                                     then
                                                        ...
                                                     else
                                                        ...)
                                                (delta^2 k)))
                                (delta^1 k)))
                (delta^0 k)))
```

An expression that appears within [and] is an intention that is expected to be true whenever control passes through the expression. It is not necessary to understand the intentions in order to understand the schema f. In fact many readers might prefer not to read the intentions. The intention functions fa, fc, and fd are intended to express what goes on in loops a, c, and d respectively. The function f can be re-written as follows:

```
(f k) = begin (register m n i j)
        (comment m, n, i, and j are registers of the program
schema f)
        (m <- k)
a:      (if (not (alpha m)) then (m <- (delta m)) (go a))
                 [define (fa m) = if (alpha m) then m else (delta
m)]
                 [(m = (fa k)) (comment It is our intent that m
be equal to (fa k) at this point.  It can be shown by induction
that this intention is always realized.)]
        (i <- k)
        (n <- (beta m))
c:      (if (alpha i)
              then
                 [(f k) = (fc (delta (fa k)) k) = n]
                 (return n))
                 [n = (f i)]
                 [define (fd n m j) = if (alpha j) then (gamma n
m) else (fd n (delta m) (delta j))]
                 [define (fc n i) = if (alpha i) then n else (fc
(fd n k i) (delta i))]
                 [n = (fd (beta (fa k)) k i)]
        (i <- (delta i))
        (j <- i)
        (m <- k)
d:      (if (not (alpha j))
              then
                 (j <- (delta j))
                 (m <- (delta m))
                 (go d))
        (n <- (gamma n m))
                 [n = (f m)]
        (go c)
        end
```

1.1.1.2 Compilation

We can look at program schemata and recursive schemata as automata that operate on the universe of terms as a data space. A finite state schema automaton operates under a finite state control structure using a finite number of registers each of which can hold one term. As a primitive operation the automaton is allowed to create a term by applying a function to terms stored in its registers and then to store the result back in a register. In addition the automaton is allowed a finite number of primitive predicates to test the contents of its registers. The class of finite state schema automata is equivalent to the class of program schemata in the obvious way. A pushdown schema automaton is defined to be a finite state schema automaton with a pushdown stack. In addition a pushdown schema automaton is allowed a finite number of distinguished constants as terms together with predicates that test for the distinguished constants. We will investigate the relationship between these machines and schemas. The appropriate kind of equivalence is one in which side effects are allowed. Two schemas will be said to be side-effect equivalent if they are the same function for all interpretations including those which involve side effects. For example the schemas j1 and j2 below are not side-effect equivalent.

```
(j1 x) = if (P x) then x
                  else (p1 (g x) (g x))

(p1 x y) = x

(j2 x) = if (P x) then x
                  else (j2 (G x))
```

The free interpretations are the ones in which each uninterpreted function symbol is interpreted as the function which evaluates to the list of all the primitive terms that have been previously evaluated in the computation. For example the side-effect protocol tree for j2 is

```
if (P x)
     then (x +(P x))
     else
        if (P (G x))
             then ((G x) +(P (G x)) (G x) -(P x))
             else
                if (P (G^2 x))
                     then ((G^2 x) +(P (G^2 x)) (G^2 x) -(P (G
x)) (G x) -(P x))
                     else...
```

On the other hand the side-effect protocol tree of j1 is:

```
if (P x)
     then (x +(P x))
     else
        if (P (G x))
             then ((G x) +(P (G x)) (G x) (G x) -(P x))
             else
                if (P (G^2 x))
                     then ((G^2 x) +(P (G^2 x)) (G^2 x) (G^2 x)
-(P (G x)) (G x) (G x) -(P x))
                     else...
```

Thus j1 and j2 are not side-effect equivalent. Although ordinary equivalence is recursively undecidable, side-effect equivalence is decidable by tree expansion.

The Compilation Theorem:

For every recursive schema there is a side-effect equivalent pushdown schema automaton: Arguments to a function are passed along by placing them on the stack and the value of the function is returned in register r1. We shall show how to compile the schema f defined below:

```
(f x) = begin (new y initial (H x))
        (comment y is a new local which is initialized to (H x))
        (if (P x)
             then (K x y)
        elseif (and y (P (f x)))
             then (K y x)
             else (G (K y x) y))
        end
```

The compiled form is

```
(f x) = begin
                (comment (= x) is the top element of stack when
f is entered.  The top element is also called (index 1).)
                (define y to be (index 2))
                (push (= x))
                (call 1 H)
                (push r1))) (comment  We declare y to be the
next element on the stack and initialize it with the value (H
x). The expression (call 1 H) will call the function H with 1
argument.)
        (if
                (push (= x))
                (call 1 P)
                (true? r1)
             then
                (push (= x))
                (push (= y))
                (call 2 K)
                (return)
        elseif
                (and
                        (
                                (true? (= y)))
                        (
                                (push (= x))
                                (call 1 f)
                                (push r1)
                                (call 1 P)
                                (true? r1)))
             then
                (push (= y))
                (push (= x))
                (call 2 K)
                (return))
        else
                (push (= y))
                (push (= x))
                (call 2 K)
                (push r1)
                (push (= y))
                (call 2 G)
                (return))
        end
```

1.1.1.3 Schemas with Resets

Tags can be thought of as identifiers which are bound at each activation level. By passing the activation as a parameter the level of activation can be immediately reset by executing a transfer of control through the activation. In order to obtain an equivalent machine, we can extend the instructions of the push down schema automaton by allowing them to store a pointer to the top element of the stack into one of the registers. The resulting class of machines is called the reset push down schema automata. If the stack is ever popped back past a location that is pointed to by a register then the automaton halts with an error. We found discussions with Mike Fischer helpful in analyzing schemas with resets.

The Reset Theorem:

The class of reset push down schema automata is equivalent to the class of ordinary push down schema automata.

We can simulate any schema that uses jumps through activations with the loss of at most a factor of two in speed. The idea is that we shall use the distinguished constants "one", "two", "three" to indicate an abnormal return. If a function returns "three" then it is doing an abnormal return on its third argument. Consider the following example:

```
(try x)  = begin
again:
        (if (Q x)
             then
                (x <- (F x))
                (go again)
        elseif (P x)
             then
                (x <- (harder (F x) again)) (comment the tag
"again" is an identifier)
                (if (not x)
                     then (return false)
                     else (go again))
             else (return false))
        end

(harder x1 tag)  = begin
again:
        (if (Q x1)
             then
                (x <- (F x1)) (comment set the global x to (F
x1))
                (go tag) (comment jump back to the activation
defined by tag)
        elseif (P x)
             then
                (x1 <- (harder (F x1) tag))
                (if (not x1)
                     then (return false)
                     else (go again)))
             else (return false)
        end
```

We can rewrite try and harder as try′ and harder′ respectively without the use of resets. We shall use the function (exit x) to force a process to leave a function with the value x.

```
(try' x)  = begin
again:
        (if (Q x)
             then
                (x <- (F x))
                (go again)
        elseif (P x)
             then
                (x <- (begin (new temp)
                        (temp <- (harder' (f x)))
                        (if (is temp "second")
                            then
                                (go again)
                            else
                                (return temp)))
                (if (not x)
                     then (return false)
                     else (go again)
            else (return false))
        end

(harder' x1)  = begin
again:
        (if (Q x1)
             then
                (x <- (F x1))
                (return ("second"))
        elseif (P x1)
             then
                (x1 <- (begin (new temp)
                        (temp <- (harder' (F x1))
                        (if (is temp "second")
                            then
                                (exit "second") (comment
"second" is the value of the function harder')
                            else
                                (return temp))))
                (if  (not x1)
                     then (return false)
                     else (go again))
          .  else (return false))
        end
```

1.1.2 Schemas with Counters

We would like to present another example of a function that can be computed by a recursive schema but not by any program schema. Define (F^n x) as in the proof of the Single Instance Theorem. Thus ((F^n+1) x) = (F (F^n x)). Suppose that we successively compute (F x), (F (F x), etc. As we successively compute the quantity (F^i x) for some integer i we shall keep a running count of the number of times that (P (F^j x)) has been true for j less than i, minus the number of times that (P (F^j x)) has been false for j less than i. If this count ever goes negative then we shall return false as the value of the function (zero x), otherwise the function (zero x) will run forever.

The Counting Theorem for Program Schemata

The recursive schema 'zero' defined below is not schematically equivalent to any program schema.

```
(zero x) = begin
        again:
                (if (P x)
                     then
                        (x <- (positive (F x)))
                        (if x
                             then
                                (go again))
                             else
                                (return false)
                     else
                        (return  false))
        end
```

The schema ‘zero’ uses the schema ‘positive’ to keep track of the count by the depth of recursion of the schema ‘positive’.

```
(positive x) = begin
        again:
                (if (P x)
                     then
                        (x <- (positive (F x)))
                        (if x
                             then
                                (go again))
                             else
                                (return false)
                     else
                        (return (F x)))
        end
```

Using the technique of loop elimination we can convert the above functions into purely recursive schemas. We shall define a schema zero1 which is equivalent to zero and a schema positive1 which is equivalent to positive.

```
(zero1 x)= (if (P x)
             then
                (if (positive1 (F x)))
                    then
                        (zero1 (positive1 (F x)))
                     else
                        false)
             else
                false)

(positive1 x)=
        (if (P x)
             then
                (if (positive1 (F x))
                     then
                        (positive1 (positive1 (F x)))
                     else
                        false)
             else
```

```
                (F x))
```

The protocol tree for the schema zero is

```
(if (P (F^0 x))
     then
        (if (P (F^1 x))
             then
                (if (P (F^2 x))
                     then
                        ...
                     else
                        (if (P (F^3 x))
                             then
                                ...
                             else
                                (if (P (f^4 x))
                                     then
                                        ...
                                     else
                                        false)))
             else
                (if (P (F^2 x))
                     then
                        (if (P (F^3 x))
                              then
                              else
                                (if (P (F^4 x))
                                     then
                                     else
                    else
                       false))
    else
       false)
```

The reader might ask himself why it is that the Single Instance Theorem doesn't apply to the schema positive1.

However a program schema can solve the problem if we give it a counter. We postulate the functions "+", "-", and zero? which respectively add, subtract, and test for zero. The following program schema is schematically equivalent the the function zero:

```
(zero1 x) = begin (new n) (return (zero2 x)) end
(zero2 x) = begin
```

```
        again: (if (P x)
                        then begin
                           (x <- (F x))
                           (n <- n + 1)
                           (go again)
                           end)
                (if (zero? n) then (return false))
                (n <- n-1)
                (go again)
        end
```

By allowing recursive schemas to use a counter, we can construct a function 'reczero' that is not equivalent to any ordinary recursive schema. the function reczero counts the number of nodes along the bottom of the L-R tree that have the property P minus the ones that do not have the property P. The function returns the value false if the count ever goes negative. We assume that arguments are evaluated from left to right.

The Counting Theorem for Recursive Schemata:

The schema (with counters) reczero defined below is not equivalent to any ordinary recursive schema.

```
(reczero x) = begin (new n) (return (reczero1 x)) end

(reczero1 x) =
        (if (BOTTOM? x)
             then
                (if (P x)
                     then begin
                        (n <- n+1)
                        (return true)
                        end
                     else begin
                        (if (zero? n) then (return false))
                        (n <- n-1)
```

```
                    (return true)
                    end
        else begin
           (if (not (reczero1 (L x))) then (return false))
           (if (not (reczero1 (R x))) then (return false))
           (return true)
           end
```

The reason that reczero is not equivalent to any recursive schema is very similar to the reason that no recursive schema can search the branches of the L-R tree in parallel. If a recursive schema is equivalent to reczero then it is constrained to search the tree in essentially the same order that reczero searches the tree. Otherwise it could be made to fall into an infinite loop on an interpretation where reczero converges. Constrained in this fashion a recursive schema has only a finite number of states in which to try to keep the count. The recursive schema cannot succeed for the same reason that no program schema is equivalent to the function zero defined above.

Conjecture: the following function is not schematically equivalent to any purely schematic recursive system of equations. The function even is supposed to test whether the number of bottom nodes of a L-R tree that are true for the predicate P is the same as the number that are false for the predicate P. The schema 'even' differs from the schema reczero in the crucial respect that 'even' always looks at all the bottom nodes before it comes to any conclusions. Thus a recursive schema that tries to imitate the schema even has a lot more room in which to maneuver. We conjecture that no recursive schema can have enough internal states (as defined in the proof of the Simultaneous Recurstion Theorem) to be equivalent to the function even defined below.

```
(even x) = begin (new n)
        (even1 x)
        (return (zero? n ))
        end

(even1 x) =
        (if (BOTTOM? x)
             then
                (if (P x)
                     then
                        (n <- n+1)
                        x)
             else
                (n <- n-1)
                x)
        else begin (even (L x)) (return (even (R x))) end
```

1.1.1.4 Decompilation

The Decompilation Theorem:

For every push down schema automaton we can effectively construct a side-effect equivalent recursive schema. We shall assume that subroutines take their arguments in registers r1, r2, etc.

Consider the following push down schema automaton:

```
(palindrome) = begin
         (comment the argument is passed in r1)
        (move x r1) (comment move the contents of r1 into the
register x)
        (call 0 end-of-stack) (comment generate the
distinguished constant "end-of-stack")
        (push r1)
pal:
```

```
(move r1 x)
(call 1 TERMIN)
if r1
     then begin
        (pop r1)
        (call 1 end-of-stack?)
        (if r1 then (return))
    again:
        (pop r1)
        (call 1 end-of-stack?)
        (if r1
             then
                (call 0 false)
                (return)
             else (go again))
        end
(move r1 x)
(call 1 P)
(if r1
     then (push r1) (go pal))
(pop r1)
(call 1 end-of-stack?)
(if r1
     then (call 0 false) (return))
(move r1 x)
(call 1 F)
(move x r1)
(go pal)
end
```

The Decompilation Theorem follows almost immediately from the Reset Theorem. The push down schema automaton palindrome is equivalent to the schema palindrome0 defined below: The function exit will force a return from a function through arbitrarily many intervening blocks. The following method for decompiling produces a program that runs in a time proportional to the time of the push down schema automaton. There is an alternative method of decompilation that involves no loss of efficiency but it is not so easy to understand because more work must be done in the translation.

```
(palindrome0 r1) = begin (new x)
        (x <- r1)
        (palindrome1 end-of-stack label1 label2)
place1:
pal:
        (r1 <- x)
        (r1 <- (TERMIN r1)
        if r1 then begin
                (halt) (comment stack under flow)
            label1:
                (r1 <- (end-of-stack?))
                (if r1 then (r1 <- (false)) (exit r1)
                    else (go again))
                end
        (r1 <- x)
        (r1 <- (P r1))
        (if r1 then (palindrome2 r1 label1 label2) (go pal))
        (halt) (comment stack under flow)
label2:
        (r1 <- (end-ofstack? r1))
        (if r1 then (r1 <- (false)) (return r1))
        (move r1 x)
        (r1 <- (F r1))
        (x <- r1)
        (go pal)
        end

(palindrome1 stacked tag1 tag2) = begin
        (go place1)
        (x <- r1)
        (palindrome1 end-of-stack label1 label2)
place1:
pal:
        (r1 <- x)
        (r1 <- (TERMIN r1)
        if r1 then begin
                (move r1 stacked)
                (go tag1)
            label1:
                (r1 <- (end-of-stack?))
                (if r1 then (r1 <- (false)) (exit r1)
                    else (go again))
                end
        (r1 <- x)
        (r1 <- (P r1))
        (if r1 then (palindrome2 r1 label1 label2) place2: (go
pal))
        (r1 <- stacked)
        (go label2)
label2:
        (r1 <- (end-ofstack? r1))
        (if r1 then (r1 <- (false)) (return r1))
        (move r1 x)
        (r1 <- (F r1))
        (x <- r1)
        (go pal)
        end (palindrome2 stacked tag1 tag2) = begin
        (go place2)
        (x <- r1)
        (palindrome1 end-of-stack label1 label2)
place1:
pal:
        (r1 <- x)
        (r1 <- (TERMIN r1)
        if r1 then begin
                (move r1 stacked)
                (go tag1)
            label1:
                (r1 <- (end-of-stack?))
                (if r1 then (r1 <- (false)) (exit r1)
                    else (go again))
                end
        (r1 <- x)
        (r1 <- (P r1))
        (if r1 then (palindrome2 r1 label1 label2) place2: (go
pal))
        (r1 <- stacked)
        (go label2)
label2:
        (r1 <- (end-ofstack? r1))
        (if r1 then (r1 <- (false)) (return r1))
        (move r1 x)
        (r1 <- (F r1))
        (x <- r1)
        (go pal)
        end
```

1.1.3 Parallel Schemas

We introduce the delimiters "<" and ">" to delimit quantities that are to be computed in parallel. Whenever a process executes an expression like <x> it divides into two processes. One process executes x and the other attempts to continue normal execution. For example in the expression <2+3>*<4*5>, the product 4*5 is computed in parallel with the sum 2+3. Thus the expression "begin <return x> (return y) end" is defined to be the value of x or y depending on which evaluates first in some particular but unspecified parallel computation. If a process executes the primitive expression "(die)" then it must commit suicide on the spot. Processes can coordinate their actions through locks. Any expression x can be locked by (lock x) provided that the expression is not already locked. If x is already locked then any process which executes (lock x) will be blocked until x is unlocked by the primitive (unlock x). However a process can execute (locked? x) which will return true is x is locked but will lock x if it is unlocked. The kind of call delimited by "<" and ">" can be implemented using the following primitives:

(create n x af) will create a new process named n with the evaluation of x which will execute the expression af if the evaluation of x is ever completed.

(resume n x) will resume the process named n with x as the value of the statement which caused control to leave n. The process which calls resume will be stopped. If the process n is already running then the process which calls resume will be blocked until n stops.

(fork n x) is exactly like (resume n x) except that the process which calls the function fork is not stopped.

We define the following function using parallel processing:

```
(f x)=(if (P x)
            then x
            else
               begin
                       <return (f (L x))>
                       (return (f (R x)))
               end)
```

The above function is determinate (ie. halts and has the same value independent of the relative speed at which the sub-processes run) on infinite binary trees in which the predicate P is true on only one node.

The Parallel Evaluation Theorem:

The function f defined is not equivalent to any recursive schema.

Proof: Suppose a set of recursive equations (f!0, f!1, ..., f!n) is schematically equivalent to f with f!0 equivalent to f. That is for all interpretations of the uninterpreted function symbols, the schemas f and f!0 are the same function. Suppose that we start up f!0 on input x and make the predicate P false for every node to which it is applied as f!0 computes along. If the computation converges then f!0 does not look at some node which is a contradiction of the supposition that f!0 is equivalant to f. Therefore the computation runs forever and the sequence of statements through which the control passes is ultimately periodic. Consider the sequence of arguments to one of the functions (call it f!i for "f subscripted by i") as the control passes through one cycle. Suppose that f!i is a function of j arguments: a!1,...,a!j. The arguments with which f!i will be called after the control has passed through one cycle are terms definable from a!1,...,a!j. Let us call them a!1^1,...,a!j^1. The situation can be diagrammed as follows:

.
.
(f!i a!1,...,a!j); the beginning of the cycle in the control structure
.
.
(f!i a!1^1, ...,a!j^1) ; We pass through the same point of the cycle in the control structure

If none of a!1^1, ...,a!j^1) is the same as one of a!1,...,a!j then we are done since the set of recursive equations is tracing j paths down an expontentially growing tree which means that some node is not looked at. If we set the interpretation so that P is true for the node then we have a contradiction. We conclude that the fact that one of a!1^1, ...,a!j^1 might be same as one of a!1,...,a!j is a nuisance. Let us call the arguments to f¦i after we have gone through the cycle k times a¦1^k,...a¦j^k. Observe that if we go through the cycle j! times then there will be some i such that i is less than j! and a¦1^i,...,a¦j^i has the property that it is an epicycle. By this we mean that some a¦q^i is the same as one of a¦1,...,a¦j if and only if it is the same as a¦q. All such a¦q do not contribute to the number of nodes examined since they are repeats of nodes previously examined in exactly the same way. The situation can be diagrammed as follows:

```
        .
        .
        (f¦i a¦1,...,a¦j);
        .
        .
        .
        (f¦i a¦1^1, ...,a¦j^1)
        .
        .
        .
        (f¦i a¦1^k, ...,a¦j^k); the beginning of the epicycle in
the control structure
        .
        .
        (f¦i a¦1^(2*k), ...,a¦j^(2*k)); we pass through the same
point in the epicycle
```

Threrefore we can complete our proof by applying to epicycles the above argument that we used for cycles. Q.E.D.

1.1.4 Locative Schemas

The Locative Theorem:

If locations of identifiers are an allowed data type, then the control structure of recursive schemas can compute any partial recursive function. We shall revert to the convention that arguments are passed on the stack. We will need to use register which we shall call r1. The register will be allowed to hold a pointer to a location in the stack. Pointers are created by the use of the function "index". For example (index 0) is the location of the top of the stack when the function is entered.

proof:

We can define a counter using a register as follows:

```
(initialize-counter1) = begin
        (call 1 zero)
        (push r1) (comment push the distinguished literal zero
onto the stack)
        (movei r1 (index 0)) (comment put a pointer to the top
of the stack in register r1)
        (return)
        end

(count-up1) = begin
        (push r1) (comment push the contents of register r1)
        (movei r1 (index 0))
        (return)
        end

(count-down1) = begin
        (move r1 (in r1)) (comment put the contents of the
contents of location r1 in location r1)
        (return)
        end

(zero-test1) = begin
        (push r1)
        (call 0 zero?) (comment test for the distinguished
literal zero)
        (return)
        end
```

Marvin Minsky proved that two counters are universal. Q.E.D.

1.1.5 Schemas with Selectors and Replacement

Another way in which we can proceed is to impose data types on the computing domain. Storage off the stack can be established by postulating a constructor c and selectors s1 and s2 such that for all x and y in the computing domain we have:

```
        (s1 (c x y)) = x
        (s2 (c x y)) = y
```

in the domain of interpretation. Classically we would postulate that every call to the constructor must return a new element of the computing domain. The lack of functionality of c in the computing domain implies that it must be defined using side effects.

1.1.6 Schmeas with Free Variables

```
(c x y) = begin (new z)
                (z <- (s1 free-storage-list))
                (free-storage-list <- (s2 free-storage-list))
                (comment "free-storage-list" is free in c)
                (return (CONSTRUCTOR x y z)
                end
```

For some purposes the hash coded constructor of McCarthy (which we shall call hc) results in gains in efficiency in both time and storage. For a hash coded constructor, we would have

if $x1=x2$ and $y1=y2$ then (hc x1 y1) = (hc x2 y2).

1.1.7 Heirarchical Schemas

PLANNER uses a more powerful control structure than that of the recursive function call. A HIeRARCHICAL CONTROL STRUCTURE is used which means that at any point a process can fail which will cause it to back up to some previous state and then continue. The primitive function (GENFAIL) will generate a simple failure. The primitive function (FAIL? try lose) will evaluate the expression try. If the evaluation succeeds then the value of the function "failure?" is the value of try. Otherwise the value of the function "failure?" is the value of lose. For example the value of

```
(+
        (failure? (x <- 2) (x <- 3))
        (if x=2 then (fail) else 4))
```

is 7 since x first gets the value 2 but then is given the value 3 when a failure backs up to the function "failure?".

1.1.7.1 Comparison with Recursive Schemas

We shall give an example to show that hierarchical control structure is more powerful than recursive control structure.

Heirarchical Schemas Are More Powerful than Recursive Schemas

The hierarchical schema g defined below is not equivalent to any recursive schema. What the schema g does is to search the following tree for x looking for a node on which P is true:

```
x
        (L^1 x)
                (L^2 x)
                        (L^3 x)
                        (R^1 (L^2 x))
                                (L^4 x)
                                (R^1 (L^4 x))
                (R^1 (L^1 x))
                        (R^2 (L^1 x)))
        (R^1 x)
                (R^2 x)
                        (R^3 x)
```

We have shown in the section on parallel schemas that no recursive schema can do the search.

```
(g x) = (h (f x))

(h z) = if (is z "true")
             then
                (begin
                        (comment if the value true is returned
then we are done)
                        true)
             else
                (begin
                        (comment otherwise generate a failure)
                        (fail))

(f x)=(fail?
        false
        (begin (new y)
                (comment y is a new local)
                (y <- x)
                (k
                        (f (L x))
                        (if (P y)
                             then true
                             else (y <- (R y) false))))

(k s t) = if (is s "true")
             then "true"
             else t
```

Proof: The proof is similar to the proof of the parallel evaluation theorem. Suppose a set of recursive equations (f#0, f#1, ..., f#n) is schematicaly equivallent to f with f#0 equivalent to f. Suppose that we start up f#0 on input x and make the prdicate p true for every node to which it is applied as f#0 computes along. If the computation converseges thn f#0 does not look at some noed which is a contradiction of the supposition that f#0 is equivalent to f. Therefore the computation runs forever and the sequence of statements therouch which the control passes is ultimately periodic.

1.1.7.2 Comparison with Parallel Schemas

We conjecture that parallel schemas are more powerful than hierarchical schemas. The example which will show this is the same example used to show that parallel schemas are more powerful than recursive schemas. However, the proof for hierarchical schemas is more difficult. The method by which parallel schemas can simulate hierarchical schemas is messy but straight forward.

1.1.7.3 PLANNER Schemas

2 Synthetic Theory

2.1 Realizations

2.1.1 Realizations for the Quantificational Calculus

We would like to show how we can use schemas to express procedurally the meaning of certain constructive logically valid sentences in the predicate calculus. Classically, intuitionistic logic has been used to prove constructive sentences. However, the connection between this language and push down schema automata is somewhat indirect. We need to define the notion of a schema g realizing a formula phi. Roughly speaking g realizes phi if it tells how to compute the value of phi from the subformulas of phi depending on the logical connectives of phi. We shall define the notion that g realizes phi by induction on the structure of phi:

For (terms). g realizes phi where phi is a term if g is true if and only if phi is true. For example (P (F w) z) realizes (P (F w) z).

For (and...). g realizes phi = (and theta psi) if (g 0) realizes theta and (g 1) realizes psi. Note that g really is two functions in disguise.

For (or...). g realizes phi = (or theta psi) if whenever (g 0) is false then (g 1) realizes psi and whenever (g 0) is not false then (g 1) realizes theta.

For (implies...). g realizes phi = (implies theta psi) if whenever h realizes theta then (g h) realizes psi.

For (not...). g realizes phi = (not theta) if for no h is it the case that h realizes theta.

For (all...). g realizes phi = (all x [theta x]) if for all x it is the case that (g x) realizes [theta x].

For (some...). g realizes phi = (some x [theta x]) if (g 1) realizes [theta (g 0)].

Consider the following formula which we shall call phi:

```
(implies
        (some x
                (implies (A x) (B x))
        (implies (all x (A x)) (some x (B x))))
```

We claim the function g defined below realizes phi.

```
g = (lambda h (lambda k (lambda s
        (if s = 0
             then (h 0)
             else ((h 1) (k (h 0)))))))

        suppose that h realizes (some x (implies (A x) (B x))
        (h 1) realizes (implies (A (h 0)) (B (h 0)))
                suppose that k realizes (all x (A x))
                (k (h 0)) realizes (A (h 0))
                ((h 1) (k (h 0))) realizes (B (h 0))
                (((g h) k) 1) realizes (B (((g h) k) 0))
                ((g h) k) realizes (some x (B x))
        (g h) realizes (implies (all x (A x)) (some x (B x)))
g realizes phi
```

we are interested in knowing when a formula can be realized constuctively.

Realization Theorem for Recursive Schemas with Functional Arguments.

If phi is proveable in intuitionistic logic, then phi is realizable by a recursive schema with functional arguments. The Realization Theorem represents one approach toward a constructive theory of computation. From a description of the kind of object that we would like to have given the description of certain other objects as input, we derive a program for computing our goal. Actually we shall prove that for intuitionistic logic the realization function can be made primitive recursive. The proof is a slight modification of the standard proof for the integers. It is a warm up for the analogous proof for the deductive system of PLANNER. However, in PLANNER we require the full power of the recursive functions for our constructive realizations.

Proof: The following proof is by induction on the structure of intuitionistic proofs using natural deduction. It goes by straightforwardly winding and unwinding of definitions. With a little work we could get PLANNER to create the proof.

```
(and introduction)
        theta realized by say g
        psi realized by say h
        ----------
        (and theta psi) realized by (lambda s (if (s = 0) then g
else h))

(and elimination)
        (and theta psi) realized by say g
        ----------
        theta realized by (g 0)
        psi realized by (g 1)

(or intro)
        psi realized by say g
        ----------
        (or theta psi) realized by (lambda t (if t=0 then false
else g
        (or psi theta) realized by (lambda t (if t=0 then true
else g)

(or elim)
        (or theta psi) realized by say g
                theta hypothesis; suppose that theta is realized
by h
                .
                .
                .
                eventually deduce say omega which is realized by
(m h) for some recursive m using the inductive hypothesis
                psi hypothesis; suppose the psi is realized by k
                .
                .
                eventually deduce omega which is realized by (l
k) for some recursive l using the inductive hypothesis
        ----------
        omega which is realized by (if (g 0) then (m (g 1)) else
(l (g 1))

(implies intro)
                omega hypothesis; suppose omega is realized by h
                .
                .
                eventually deduce say psi which is realized by
(g h) for some recursive g using the inductive hypothesis.
        ----------
        (implies omega psi) realized by (lambda h (g h))
```

```
(implies elim)
        (implies omega psi) realized by say g
        omega realized by say h
        ----------
        psi realised by (g h)

(neg intro)
                omega hypothesis; suppose that omega is realized
by h
                .
                .
                eventually deduce say (not psi) which is
realized by (g h) for some recursive g using the inductive
hypothesis
                eventually deduce psi which is realized by (k h)
for some recursive k using the inductive hypothesis.
        ----------
        (not omega) which is realized by any function since it
is impossible for both (not psi) to be realized by (g h) and for
psi to be realized by (k h).

(all intro)
                x¦
                 ¦
                .¦
                .
                 ¦
                 ¦eventually deduce say [omega x] which is
realized by (g x) for some recursive g using the inductive
hypothesis
        ----------
        (all x [omega x]) realized by (lambda x (g x))

(all elim)
        (all x [omega x]) realized by say g
        ----------
        [omega t] for some term t; realized by (g t)

(exist intro)
        [omega t] is realized by say g where t is a term
        ----------
        (exist x [omega x]) is realized by (lambda s (if (s = 0)
then t else g))

(exist elim)
        (some x [omega x]) realized by say g
                x¦[omega x] realized by (g 1)
```

```
                    :
                    :
                   .
                   .
                   .:eventually deduce say psy which does not
contain x free:  psy is realized by (m (g 0) (g 1)) for some
recursive m using the inductive hypothesis.
        ----------
        psy
```

Thus we have completed the inductive proof.

Intuitionistic Implementation Theorem

For every recursive schema P, we can effectively find a first order formula [theta x y] such that P is total recursive if and only if (all x (some y [theta x y])) is proveable in intuitionistic logic. Furthermore, the program P on input x converges to the value y if and only if [theta x y] is proveable in intuitionistic logic. We assume that all uninterpreted function symbols in schemas are total.

We shall give an example of how to construct the formula theta for the following program which is due to Paterson:

```
(g x) = (if (T (F x))
             then (h x (F x))
             else x)

(h x y)=
        (if (T (F (F y)))
             then x
        elseif (T (F x))
             then (h (F x) (F (F y)))
        else (g (F x)))
```

The formula [theta x y] to be constructed is the conjunction of the following three formulas where "iff" is an abbreviation for "if and only if":

```
(iff
        (PG x y)
        (or
                (and (T (F x)) (PH x (F x) y))
                (and (not (T (F x)) (y = x))))

(all x1 x2 y (iff
        (PH x1 x2 y)
        (or
                (and (T (F (F x2))) (y = x1))
                (and
                        (not (T (F (F x2)))
                        (T (F x1))
                        (PH (F x1) (F (F x2)) y))
                (and
                        (not (T (F (F x2)))
                        (not (T (F x1))
                        (PG (F x1) y)))

(all x (or (T x) (not (T x))))
```

The last statement comes from the fact that we are assuming that all uninterpreted functions are total. The schema g is indeed total recursive.

Even after adding selectors and constructors the realization theorem can still be proved in the standard way. We introduce the predicate atom which tests to see if its argument is atomic and thus cannot be broken down using the selectors. The following rule is added to intuitionistic logic:

```
        (all x (implies (atom x) [theta x])) realized by say g
                x,y|[theta x] hypothesis; suppose [theta x] is
realized by (m x)
                   |[theta y] hypothesis; [theta y] is realized
by (m y)
```

```
                   :
                .
                .
                .  :
                   :eventually deduce [theta (c x y)] realized
by say (h m x y) using the inductive hypothesis
        ----------
        (all x [theta x]) realized by
(k x) = (if (atom x)

             then (g x)

             else (h k (s1 z) (s2 z)))
```

Sometimes an increase in efficiency can be obtained from replacement operators r1 and r2 such that

if x = (s1 z) and y = (s2 z) then after (r1 z w) we have (s1 z) = w, and (s2 z) = y
if x = (s1 z) and y = (s2 z) then after (r2 z w) we have (s1 z) = x, and (s2 z) = w.

There are real problems in trying to use the universe of terms as a universal domain of interpretation when the use of replacement operators is allowed.

2.1.2 Realizations of PLANNER Theorems

2.2 Construction of Schemas

2.2.1 Completeness of Procedural Abstration

2.2.2 Completeness of Method of Canned Loops

3 Current Problems and Future Work

How can we characterize more precisely the difference between functions that need to use a recursive or parallel control structure as opposed to those that only need a simple iterative program structure? The problem of deciding whether any given recursive schema can be rewritten as a program schema is of course undecidable. We would like to find general criteria of independent interest which would be sufficient to guarantee that a recursive schema could not be rewritten as a program schema.

There is general agreement that the theory of computation is currently not in good shape. The three major areas (automata theory, recursive function theory, and special case hacks) are not applicable to practical programs. We can contrast our plight with the situation in applied physics. An applied physicist finds that it is essential to understand fundamental physical laws both in designing his experiments and in interpreting their results. No such fundamental laws and principles are known in programming. Recursive function theory sets the very outer limits of what is possible. Few theories are more elegant. However, the fact that classical recursive function theory deals with the indices of the partial recursive functions and not with the meaning of the programs has been a fundamental limitiation on the applicability of the theory. For example the recursion theorem says that fixed points exist for any acceptable Goedel numbering. Almost all the classical theorems of recursive function theory can be derived using only the Godel axioms for indices of partial recursive functions. Similarly, the complexity theory of the recursive functions can be derived from Blum's axioms for indices. Automata theorists have been able to discover some of the structure of various limited classes of automata such as finite state machines, push down machines, and space and time bounded machines. However, since the theory developed has been mostly concerned with closure and complexity properties of the special machines considered as acceptors, it has had limited applicability to real computer programs. Most programs are not structured in the way required to fall into one of the special classes of machines. Some theorists hope that by studying enough examples of very narrow domains of algorithms where we have a lot of domain dependent knowledge that we can induct a theory of computation in a Baconian fashion. Deep studies have been made on questions such as how fast integers can be multiplied and how fast matrices can be multiplied. Studies in the theory of searching and sorting appear to be more relevant for constructing a unified theory of computation since they are concerned with basic computational abilities.

Studying the properties of programs schematically offers several advantages. Schemas can be programmedi in a realistic fashion. They mirror the structure of programs that are used in applications. Using them we can precisely define structural properties. Properties of the structural classes can be demonstrated. An encouraging sign is that none of the proofs given here is conceptually very difficult. Schemas give us a tool by which we can rigorously formulate and prove statements that every programmer intuitively knows. We have used schemas to make a kind of distinction between semantic and syntactic extensions to programming languages. The intent of the restriction that functions be uninterpreted is to try to prevent our mathematics from falling into what Perlis likes to call the "Turing Machine Tar Pit." By using uninterpreted function symbols we can prove both analytic and constructive theorems about classes of programs. In the analytic theory the mathematical properties of the structural classes is expounded. In the constructive theory the process by which schemas can be constructed from goal oriented language such as PLANNER. The intention is only partially realized and we must search for other natural mathematical structures to impose on our schemas in order to obtain a more realistic theory of semantic extensions to programming languages. We are continuing to investigate what gains in efficiency can be obtained from the following extensions to programming languages:

recursion

hierarchical control structure

PLANNER primitives

Locations as a type

resets

free identifiers

parallel evaluation

replacement operators for constructors.