Working Paper 164

May 1978

Using Message Passing Instead of the GOTO Construct

Carl Hewitt

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

 DRAFT April 1978

Using Message Passing Instead of the GOTO Construct

Carl Hewitt

M.I.T.

Room 813
545 Technology Square
Cambridge, Mass. 02139
(617) 253-5873

SECTION I --- ABSTRACT

This paper advocates a programming methodology using message passing. Efficient programs are derived for fast exponentiation, merging ordered sequences, and path existence determination in a directed graph. The problems have been proposed by John Reynolds as interesting ones to investigate because they illustrate significant issues in programming. The methodology advocated here is directed toward the production of programs that are intended to execute efficiently in a computing environment with many processors. The absence of the GOTO construct does not seem to be constricting in any respect in the development of efficient programs using the programming methodology advocated here.

SECTION II --- INTRODUCTION

The programming problems arising from use of the GOTO construct have become well known over the last decade. However, removal of the construct from traditional programming languages has been found by many to be constricting and to result in inefficient programs. A number of proposals have been advanced to allow the construct in a restricted way [e.g. Knuth: 1974 and Reynolds: 1977].

For the past several years I have been investigating the message passing metaphor of computation and have found that it can be used to conveniently and efficiently implement iterative programs. PLASMA [Hewitt and Smith: 1975] seems to have been the first concurrent programming language with no special primitive control constructs for iteration. Iteration emerges as one of the patterns of passing messages that is inherent in the basic structure of PLASMA. The underlying message passing semantics of the language [Greif and Hewitt: 1975, Hewitt 1977, Hewitt and Baker: 1977, Hewitt and Attardi: 1978] led directly to the development of this paradigm for iteration. Before the development of PLASMA the general view [Allen and Cocke: 1972; Paterson and Hewitt: 1970; Strong: 1971] was that certain recursive procedures [sometimes called "tail-recursive"] could be recognized by the compiler and "translated" into iterative loops using the GOTO construct. On a message passing machine in a language like PLASMA, no such "recognition" and "translation" is necessary.

In this paper three problems for which Reynolds [1977] has developed implementations using the GOTO construct are examined. Using the programming methodology advocated here, I did not find the absence of the GOTO construct to be a limitation in the development of efficient implementations for any of the problems. However, this does not say that Reynolds [1977] and Knuth [1974] are incorrect in their claim that the absence of the GOTO construct can be constricting in a more traditional programming methodology. In my view the fundamental limitation of the GOTO construct is that it provides for the transfer of control to the target site of the GOTO without allowing for the possibility of sending along a message with the transfer of control. Thus the only way to provide information to the target site of a GOTO is to use a side effect on some resource shared with the target site.

SECTION III --- FAST EXPONENTIATION

The first problem that will be investigated is to efficiently compute x^n. I.e. the goal is to develop an program called **exponentiate** satisfying the following specification:

$$\textit{if } (n \geq 0) \textit{ then } (\text{exponentiate } x\ n) = x^n$$

To synthesize this program notice the following facts:

$$(\text{exponentiate } x\ 0) = 1$$

$$\textit{if } (n > 0) \textit{ then } (\text{exponentiate } x\ n) = (x * (\text{exponentiate } x\ (n - 1)))$$

In the implementation, use will be made of a *rules* expression of the following form:

```
(rules expression
   (pattern1 → body1)
   ...
   (patternn → bodyn)
 else
   bodyn+1)
```

such that if the value of expression matches any of the pattern_i then the corresponding body_i is executed else body_{n+1} is executed. If the value of expression matches more than one of the pattern_i then an arbitrary one of the corresponding body_i is selected to be executed. The *rules* expression is related to the guarded commands of Dijkstra [1975] but is derived from an independent source: the PLANNER-like pattern directed programming languages [PLANNER, QA-4, POPLER, CONNIVER, PLASMA, etc.].

Putting these facts together, the following implementation is derived:

```
(exponentiate =x =n) ≡      ;x^n is computed as follows
    (rules n      ;the rules for n are
       (0 → 1)      ;if it is 0 then the value is 1
     else
       (x * (exponentiate x (n - 1))))      ;else the value is x times the value of x^n-1
```

Note the following two additional facts that can help us to optimize the implementation:

(exponentiate x 1) = x

if n *is even, then* (exponentiate x n) = (exponentiate (x * x) (n / 2))

Refining the implementation using the above facts produces:

```
(exponentiate =x =n) ≡          ;x^n is computed as follows
    (rules n        ;the rules for n are
       (0 → 1)      ;if n is 0 then the value is 1
       (1 → x)      ;if n is 1 then the value is x
       ((∧ (¬ 0) (even)) →      ;if n is nonzero and even
                    (exponentiate (x * x) (n / 2)))     ;then the value is (x*x)^(n/2)
     else
       (x * (exponentiate x (n - 1))))      ;else the value is x * x^(n-1)
```

In the evaluation of the *else* clause of exponentiate, it is known that n is greater that 1 and that it is not even which implies that (n - 1) is non-zero and even. Therefore the expression

(exponentiate x (n - 1))

will result in redundant testing on the next invocation of exponentiate. However, the above expression can be expanded to

(exponentiate (x * x) ((n - 1) / 2))

producing the following implementation which does not do any redundant testing:

```
(exponentiate =x =n) ≡          ;x^n is computed as follows
    (rules n        ;the rules for n are
       (0 → 1)      ;if n is 0 then the value is 1
       (1 → x)      ;if n is 1 then the value is x
       ((∧ (¬ 0) (even)) →      ;if n is nonzero and even
                    (exponentiate (x * x) (n / 2)))     ;then the value is (x*x)^(n/2)
     else
       (x * (exponentiate (x * x) ((n - 1) / 2))))      ;else the value is x*(x*x)^((n-1)/2)
```

The above implementation is inefficient because it uses extra storage in the recursive invocations of itself. Applying a standard transformation [Standish et. al.: 1976, Burstall and Darlington: 1977, Strong: 1971] for translating recursive procedures into iterative procedures, produces the following implementation:

```
(exponentiate =x =n) ≡        ;to compute x^n
    (rules n      ;the rules for n are
        (0 → 1)      ;if n is 0 then the value is 1
    else      ;else the value is given by
        (exponentiate_loop x n 1))      ;(exponentiate_loop x n 1) which is defined below
```

where

```
(exponentiate_loop =base =index =accumulation) ≡
;the iterative procedure exponentiate_loop is implemented as follows
    (rules index
        (1 → (accumulation * base))
        ((even) → (exponentiate_loop (base * base) (index / 2) accumulation))
      else
        (exponentiate_loop (base * base)  ((index - 1) / 2) (base * accumulation)))
```

The procedure exponentiate_loop satisfies the following specification:

if index>0 *then* (exponentiate_loop base index accumulation) = accumulation * $base^{index}$

The above implementation was derived in consultation with Bill Ackerman. It has no "redundant testing". The message passing aspects of this example are discussed in "Viewing Control Structures as Patterns of Passing Messages".

SECTION IV --- MERGING TWO SORTED SEQUENCES

The essential aspect of the second problem is to merge two sorted sequences x and y to produce a third sorted sequence which has all the elements of x and y. The implementation of merge makes use of the unpack operator [which is a unary prefix operator denoted by "!"] on sequences. For example if x is the sequence [3 4] and y is the sequence [9 8] then the following equivalences hold:

```
[x !y]   =  [[3 4] ![9 8]]  =  [[3 4] 9 8]
[!x !y]  =  [![3 4] ![9 8]]  =  [3 4 9 8]
[!x 5 !x]  =  [![3 4] 5 ![3 4]]  =  [3 4 5 3 4]
```

The unpack operator is also used in pattern matching where it is restricted in use to matching the remainder of a sequence:

if the pattern [=x !=y] *is matched against* [1 2 3] *then*
x *is bound to* 1 *and*
y *is bound to* [2 3]

if the pattern [4 =x !=y] *is matched against* [4 5] *then*
x *is bound to* 5 *and*
y *is bound to* []

if the pattern [=x [4 !=y] !=z] *is matched against* [9 [4 3] 7 8 9] *then*
x *is bound to* 9
y *is bound to* [3] *and*
z *is bound to* [7 8 9]

```
(merge =x =y) ≡      ;to merge two sorted sequences x and y
   (rules [x y]      ;the rules for x and y are
     ([x []] → x)      ;if y is an empty sequence, the value is x
     ([[] y] → y)      ;if x is an empty sequence, the value is y
     ([[=first_x !=rest_x] [=first_y !=rest_y]] →
     ;otherwise let first_x be the first element of x
     ;rest_x be the rest of the elements of x
     ;first_y be the first element of y
     ;rest_y be the rest of the elements of y
          (if (first_x ≤ first_y)      ;if first_x  first_y
            then      ;then the value is
               [first_x !(merge rest_x y)]      ;the sequence of first_x followed by the merge of rest_x and y
            else      ;else the value is
               [first_y !(merge x rest_y)])))
               ;the sequence of first_y followed by the merge of x and rest_y
```

It is easy to prove that the above implementation meets the following specifications:

if (sorted x) ∧ (sorted y) *then* (sorted (merge x y))

(elements (merge x y)) = ((elements x) ∪ (elements y))

where (elements z) is the multi-set of all the elements of the sequence z with duplicates preserved and the ∪ operation likewise preserves duplicates. Using the obvious recursive definition of sorted for sequences and elements and = for multi-sets, the proof that the above implementation meets the specifications can probably be automated [Boyer and Moore: 1977; Luckham: 1977; Good, London, and Bledsoe: 1975; Waldinger and Levitt: 1974; Deutsch: 1973; and Rich, Shrobe, Waters, Sussman, and Hewitt: 1978].

The above implemeptation of merge is analogous to the first implementation of exponentiate in that it does redundant testing. The recursive call (merge rest_x y) redundantly tests whether or not y is empty and the recursive call (merge x rest_y) redundantly tests whether or not x is empty.

Therefore define a new procedure (special_merge first_s1 rest_s1 s2) which merges first_s1, a sequence rest_s1, and a sequence s2 as follows:

```
(special_merge =first_s1 =rest_s1 =s2) ≡
    (rules s2
        ([] → [first_s1 !rest_s1])
        ([=first_s2 !=rest_s2] →
                    (if (first_s1 ≤ first_s2)
                        then [first_s1 !(special_merge first_s2 rest_s2 rest_s1)]
                        else [first_s2 !(special_merge first_s1 rest_s1 rest_s2)])))
```

where special_merge satisfies the following specification:

```
if (sorted s2) ∧ (sorted [first_s1 !rest_s1]) then
                    (special_merge first_s1 rest_s1 s2) = (merge [first_s1 !rest_s1] s2)
```

Using special_merge the implementation of the original merge function can be refined to be the following:

```
(merge =x =y) ≡
   (rules [x y]
     ([x []] → x)
     ([[] y] → y)
     ([[=first_x !=rest_x] [=first_y !=rest_y]] →
                                  (if (first_x ≤ first_y)
                                      then [first_x !(special_merge first_y rest_y rest_x)]
                                      else [first_y !(special_merge first_x rest_x rest_y)])))
```

Again using the standard transformation for translating a recursive procedure into an iterative procedure, special_merge can be written iteratively as follows:

```
(special_merge =x =s1 =s2) ≡ (special_merge_loop x s1 s2 [])

where

(special_merge_loop =first_s1 =rest_s1 =s2 =accumulation) ≡
      (rules s2
          ([] → [!accumulation first_s1 !rest_s1])
          ([=first_s2 !=rest_s2] →
                              (if (first_s1 ≤ first_s2)
                                  then (special_merge_loop first_s2 rest_s2 rest_s1 [!accumulation first_s1])
                                  else (special_merge_loop first_s1 rest_s1 rest_s2 [!accumulation first_s2]))))
```

where special_merge_loop satisfies the following specification:

```
if (sorted s2) ∧ (sorted [first_s1 !rest_s1]) then
      (special_merge_loop first_s1 rest_s1 s2 accumulation) = [!accumulation !(merge [first_s1 !rest_s1] s2)]
```

This produces an iterative implementation which does no redundant testing.

SECTION V --- Path Existence Determination

The final problem is to write a procedure which will determine whether or not a path (of length zero or greater) exists in a directed graph from some member of a set [without duplicates] of nodes X to a member of a set of nodes Y. The implementor is provided with two procedures such that (immediate_successors Z) is a set of the immediate successors of a node set Z and (immediate_predecessors Z) is a set of the immediate predecessors of Z. In this section we will use U to denote the ordinary set theoretic union of sets without duplicates. These procedures satisfy the following properties:

```
if (node n) ∧ (node m) then
            (m ∈ (immediate_predecessors {n})) if and only if (n ∈ (immediate_successors {m}))

if (node_set Z) then
            (immediate_successors Z) = U_{n∈Z} (immediate_successors {n})

if (node_set Z) then
            (immediate_predecessors Z) = U_{n∈Z} (immediate_predecessors {n})
```

Below an implementation is defined such that invoking an expression of the form (*path_exists_from* X *to* Y) returns either true or false depending on whether or not there is a path from X to Y of length zero or greater.

First implement a data structure called a growth that records how much of the graph has been explored. An expression of the form (create_growth X Y) will create an actor which accepts messages of the form (*insert_as_predecessor_Y* (*node*: n)) to insert a node n as a predecessor of Y and messages of the form (*insert_as_successor_X* (*node*: n)) to insert a node n as a successor of X. In both cases the growth either performs the insertion or reports that the node is already known to be a successor of X or a predecessor of Y. An expression of the form (one_at_a_time r) creates an actor which only allows one actor at a time access through it to the actor r. The properties of one_at_a_time serializers are analyzed in [Yonezawa: 1977] and [Atkinson and Hewitt: 1978].

In the implementation below, use will be made of a *cases* expression of the following form:

(*cases*
  (pattern$_1$ → body$_1$)
  ...
  (pattern$_n$ → body$_n$))

which when evaluated creates an actor with the behavior that if it receives a message which matches any of the pattern$_i$ then the corresponding body$_i$ is executed and its value is returned as the reply to the message. If the created actor receives a message that matches more than one of the pattern$_i$ then an arbitrary one of the corresponding body$_i$ is selected to be executed. If a message received does not match any of the pattern$_i$ then an error is signaled.

The procedure create_growth defined below creates a growth data structure that records how much of the graph has been examined. Evaluating an expression of the form (create_growth X Y) will produce an actor (created by one-at-a-time with one acquaintance which is an actor created by evaluating the *cases* expression below) which has two acquaintances which are cells called PREDECESSORS_Y and SUCCESSORS_X whose initial contents are Y and X respectively.

```
(create_growth =X =Y) ≡
;to create a new growth for node sets X and Y
   (let
      {(PREDECESSORS_Y initially Y)
      ;let PREDECESSORS_Y initially be Y
        (SUCCESSORS_X initially X)}
        ;and the SUCCESSORS_X initially be X
      (one_at_a_time
      ;only one activity at a time is allowed in the actor created by the cases expression below
            (cases
            ;the cases for messages received are as follows
               ((insert_as_predecessor_Y (node: =n)) →
               ;if the message received is a request to insert a node n as a predecessor of Y
                  (rules n
                  ;then the rules for n are
                     ((∈ PREDECESSORS_Y) →
                     ;if n is already an element of PREDECESSORS_Y
                        "already_present_as_predecessor_Y")
                        ;then this is reported
                     ((∈ SUCCESSORS_X) →
                     ;if n is already an element of SUCCESSORS_X
                        "already-present_as_successor_X")
                        ;then this is reported
                   else
                     (PREDECESSORS_Y ← (PREDECESSORS_Y U {n}))
                      ;else the node n is added to PREDECESSORS_Y
                      "not_present"))
                      ;and the fact that it is not already present in the growth is reported
               ((insert_as_successor_X (node: =n)) →
                  (rules n
                     ((∈ SUCCESSORS_X) →
                        "already_present_as_successor_X")
                     ((∈ PREDECESSORS_Y) →
                        "already-present_as_predecessor_Y")
                   else
                     (SUCCESSORS_X ← (SUCCESSORS_X U {n}))
                     "not_present")))))
```

Note that it is trivial to establish the following invariants for the above module:

$$X \subseteq SUCCESSORS\_X$$
$$Y \subseteq PREDECESSORS\_Y$$
$$\textit{if } (X \cap Y)=\{\} \textit{ then } (PREDECESSORS\_Y \cap SUCCESSORS\_X)=\{\}$$

The program below uses the *either* construct [called "*amb*" in McCarthy: 1963, see also Ward: 1974] so that the value of an expression of the form (*either* E_1 E_2) is computed by evaluating E_1 and E_2 in parallel and selecting the value of whichever one finishes first. When one of these two computations produces a reply, work on the other one is terminated. We only use the *either* construct in contexts where E_1 and E_2 produce the same value when both produce values.

The implementation also makes use of the *parallel_or* construct [Manna and McCarthy 1970, Paterson and Hewitt 1971] so that evaluation of an expression of the form (*parallel_or* E_1 E_2) begins computing the values of E_1 and E_2 in parallel. If either evaluation produces the value **true** then the other computation is terminated and the value of the whole expression is **true**. If both evaluations produce **false** then the value of the whole expression is **false**. Otherwise the computation continues.

In the implementation below use will be made of expressions of the form (*send* m *to* t) to send a message actor m to a target actor t.

Finally the implementation makes use of the unpack operator for sets. For example if the pattern {=x !=y} is matched against the set {3} then x is bound to 3 and y is bound to {}. Furthermore the match is nondeterministic in the sense that if the pattern {=u !=v} is matched against the set {5 6} then either u is bound to 5 and v is bound to {6} or u is bound to 6 and v is bound to {5}.

```
(path_exists_from =X to =Y) ≡        ;to determine if a path exists from a node set X to a node set Y
   (if ((X ∩ Y) ≠ {})        ;if X intersect Y is nonempty
      then true      ;then the value is true
      else      ;else
         (let {(g = (create_growth X Y))}      ;create a new growth g for X and Y
            (either      ;compute the following two expressions in parallel and return the value of either
                  (successors (immediate_successors X) reach_Y)
                  ;determine whether or not the immediate successors of X reach Y
                  (predecessors (immediate_predecessors Y) reach_X))
                  ;determine whether or not the immediate predecessors of Y reach X
            where
            (successors =S reach_Y) ≡
            ;to determine whether or not the successors of a node set S reach Y
               (rules S      ;the rules for node set S are
                  ({} → false)      ;if it is the empty set then the value is false
                  ({=an_element_S ⁞=REST_S} →
                  ;select an element of S and call it an_element_S
                  ;let REST_S be (S - {an_element_S})
                     (parallel_or      ;take the parallel_or of the following two expressions
                        (successors REST_S reach_Y)
                        ;determine whether or not the successors of REST_S reach Y
                        (rules (send (insert_as_successor_X (node: an_element_S)) to g)
                        ;the rules for the result of asking g to insert an_element_S as a successor of X are
                           ("already_present_as_successor_X" → false)
                           ;if an_element_S is already present as a successor of X then the value is false
                           ("already_present_as_predecessor_Y" → true)
                           ;if an_element_S is already present as a predecessor of Y then the value is true
                           ("not_present" →
                           ;if an_element_S is not present in the growth g
                              (successors (successors {an_element_S}) reach_Y))))))
                           ;then the value it is determined by whether or not its successors reach Y
            and
            (predecessors =S reach_X) ≡
               (rules S      .
                  ({} → false)
                  ({=an_element_S ⁞=REST_S} →
                     (parallel_or
                        (predecessors REST_S reach_X)
                        (rules (send (insert_as_predecessor_Y (node: an_element_S)) to g)
                           ("already_present_as_predecessor_Y" → false)
                           ("already_present_as_successor_X" → true)
                           ("not_present" →
                              (predecessors (predecessors {an_element_S}) reach_X))))))))
```

I realize that the growth actor g created in the above implementation will probably be a serious bottleneck in a multiprocessor system. The next version of this paper will contain a further refinement of the growth abstraction that alleviates this bottleneck.

Mathematically the successors and predecessors of a node set W can be defined as follows:

$$(\text{successors } W) \equiv \bigcup_{i \in N} (\text{immediate\_successors}^i\ W)$$

$$(\text{predecessors } W) \equiv \bigcup_{i \in N} (\text{immediate\_predecessors}^i\ W)$$

where N is the set of non-negative integers and for any function f the notation f^i denotes the i-fold composition of f with itself. I.e. for any function f, f^0 is the identity function and for any non-negative integer i, $(f^{i+1}\ x)$ is $(f\ (f^i\ x))$. The problem to be solved can be formally specified as follows:

(path_exists_from X *to* Y)=true *if and only if* ((successors X) ∩ (predecessors Y))≠{}

if (*path_exists_from* X *to* Y)=false, *then* ((successors X) ∩ (predecessors Y))={}

It is not difficult to demonstrate informally that the implementation meets the above specifications. First note that the following mathematical equivalences hold:

((successors X) ∩ (predecessors Y))≠{} *if and only if* (X ∩ (predecessors Y))≠{}
((successors X) ∩ (predecessors Y))≠{} *if and only if* ((successors X) ∩ Y)≠{}

Furthermore the following properties hold for (*successors ... reach_Y*) and (*predecessors ... reach_X*).

(*successors* X *reach_Y*)=true *if and only if* ((successors X) ∩ Y)≠{}
(*predecessors* Y *reach_X*)=true *if and only if* (X ∩ (predecessors Y))≠{}
if (*successors* X *reach_Y*)=false, *then* ((successors X) ∩ Y)={}
if (*predecessors* Y *reach_X*)=false, *then* (X ∩ (predecessors Y))={}
if (X ∩ Y)={}, *then* (*predecessors* Y *reach_X*)=(*predecessors* (immediate_predecessors Y) *reach_X*)
if (X ∩ Y)={}, *then* (*successors* X *reach_Y*)=(*successors* (immediate_successors X) *reach_Y*)

It is trivial to prove that the implementation satisfies the specifications if (X ∩ Y)≠{}. On the other hand if (X ∩ Y)={}, then the above properties together with the specifications of the *either* primitive are sufficient to establish the result.

Work is underway to formalize the above proof using the methods of Hewitt and Attardi [1978].

SECTION VI --- CONCLUSIONS

In "Viewing Control Structures as Patterns of Passing Messages" there is a sketch of a proof that any concurrent program which uses the GOTO construct [e.g to jump out of the middle of blocks] can be effectively transformed without any significant loss of efficiency into one which uses only ordinary message passing [see also McCarthy: 1960 and Steele: 1977]. Theoretically this settles the question of whether or not the absence of the GOTO is constraining in some absolute sense. However, in practice, I do not recommend that this transformation be used. Instead programs should be synthesized by progressive refinement of their specifications and implementations as has been attempted in this paper.

The result that programs can be written efficiently without use of the GOTO construct does not in and of itself imply that programs can be written efficiently without use of assignment commands. Greif and Hewitt [1975] have developed a procedure for effectively transforming any *sequential* program which uses assignment commands into an equivalent program which does not have any assignments. However, it is not known whether or not efficiency bounds which they derive are the best that can be obtained for typical sequential algorithms. Assignment commands are a significant source of difficulty in concurrent programs. They can limit the amount of concurrency that is possible. Furthermore they can easily be a source of subtle bugs because of race conditions. It is interesting to note that the programs derived in this paper for fast exponentiation and merging sorted sequences do not make use of assignment commands yet are essentially as efficient as the corresponding programs derived by Reynolds using GOTO and assignment commands. In fact the only use for assignment commands in the programs considered in this paper is to implement communication between independent concurrent processes. These assignment commands are only used inside resources protected by serializers [Hewitt and Atkinson: 1975] which carefully schedule access to the resources which they are protecting. It is currently an open research question what the important uses of assignment commands are *in a system with a large number of processors connected by a high-bandwidth packet-switched network* beyond implementing communication between independent concurrent processes.

Because of the development of extremely large scale integrated circuit chips and geographically distributed computer networks, it appears that at some point in the not too distant future almost all programs will execute in a multiprocessor environment. Thus we should design our programming languages and algorithms to be appropriate for this new environment.

SECTION VII --- ACKNOWLEDGEMENTS

The comments and criticisms of Bill Ackerman, Russ Atkinson, Gerry Barber, Roger Duffey, Henry Lieberman, Kenneth Kahn, and Guy Steele have materially improved the presentation and content of this paper. The examples which are treated in this paper were proposed as interesting ones to investigate by [Reynolds 1977]. Conversations with John Reynolds, Robin Milner, and Rod Burstall at an Edinburgh outing in July 1977 provided the impetus for the development of the ideas in this paper.

I was inspired to investigate the message passing paradigm for programming languages by a lecture which Alan Kay delivered at the MIT Artificial Intelligence Laboratory in November 1972 on a

preliminary version of the SMALLTALK-72 language [Shoch: 1977] which has subsequently evolved into SMALLTALK-76 [Kay: 1977 and Ingalls: 1978]. The SMALLTALK work in turn builds on SIMULA [Birtwistle et. al.: 1973]. This paper and companion papers [Baker and Hewitt: 1977, Hewitt and Atkinson: 1977, Hewitt and Attardi: 1978] attempt to extend this programming language paradigm into the realm of concurrent programming.

The attempt by Sussman and Steele to relate PLASMA and the actor message passing model of computation to the lambda calculus influenced them to abandon dynamic [fluid] scoping in favor of lexical scoping so that their language SCHEME [Steele and Sussman: 1978] could also use the paradigm for iteration developed for PLASMA. Steele [Steele 1977] has further developed the paradigm in the form of a working compiler for SCHEME.

SECTION VIII --- BIBLIOGRAPHY

Atkinson R. and Hewitt, C. "Specification and Proof Techniques for Serializers" IEEE Transactions on Software Engineering. 1978. To appear.

Boyer, R. S. and Moore, J. S. "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory". IJCAI-77. Cambridge. August 1977. pp 511-519.

Baker, H. J. Jr. and Hewitt, Carl "Incremental Garbage Collection of Processes" MIT A.I. Memo 454. December, 1977.

Deutsch, L. P. "An Interactive Program Verifier" Xerox Palo Alto Research Center. Report No. CSL-73-1. May, 1973.

Dijkstra, E. W. "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs" CACM. Vol. 18. No. 8. August 1975. pp 453-457.

Good,D. I.; London, R. L.; and Bledsoe, W. W. "An Interactive Verification System" IEEE Transactions on Softwared Engineering. Vol. 1. 1975. pp 59-67.

Greif, I. "Semantics of Communicating Parallel Processes" MAC Technical Report TR-154. September 1975.

Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73" Proceedings of ACM SIGPLAN-SIGACT Conference. Palo Alto, California. January, 1975.

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages" A.I. Journal. Vol. 8. No. 3. June 1977. pp. 323-364.

Hewitt, C. and Atkinson, R. "Synchronization in Actor Systems" Proceedings of Conference on Principles of Programming Languages. January 1977. Los Angeles, Calif.

Hewitt, C. and Attardi, G. "An Axiomatic Denotation Specification of a Concurrent Programming Language" MIT Working Paper. May 1978.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, 1. March 1975. pp. 26-45.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" IFIP-77. Toronto. pp 987-992.

Hewitt, C. and Baker, H. "Actors and Continuous Functionals" IFIP Working Conference on Formal Description of Programming Concepts" August 1-5, 1977. St. Andrews, New Brunswick, Canada. MIT A. I. Memo 436A. MIT/LCS Technical Report 194. December 1977.

Knuth, D. E. "Structured Programming with GO TO Statements" Computing Surveys. December 1974.

Luckham, D. C. "Program Verification and Verification-Oriented Programming" Invited paper at IFIP-77. Montreal. August 1977. pp 783-793.

Ingalls, D. H. H. "The Smalltalk-76 Programming System Design and Implementation" Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. January 23-25, 1978. Tucson, Arizona. pp. 9-16.

Manna, Z. and McCarthy, J. "Properties of Programs and Partial Function Logic" Machine Intelligence 5. Edinburgh University Press. 1970.

McCarthy, J. "Recursive Functions of Symbolic Expressions and their Computation by Machine - I" CACM. Vol 3. No. 4. April 1960. pp 184-195.

McCarthy, J. "A Basis for a Mathematical Theory of Computation" in Computer Programming in Formal Systems P. Braffort and D. Hirschberg (eds.). North Holland. 1963. pp. 33-70.

Kay,A. "Microelectronics and the Personal Computer" Scientific American, September 1977.

Paterson, M. S. and Hewitt, C. E. "Comparative Schematology" ACM Conference Record of Working Conference on Concurrent Systems and Parallel Computation. 1970. Available from ACM.

Reynolds, J. C. "Programming with Transition Diagrams" Report DAI-37. Department of Artificial Intelligence, University of Edinburgh. Also Report CSR-4-77. Department of Computer Science, University of Edinburgh. July, 1977.

Rich,C.; Shrobe, H. E.; Waters, R. C.; Sussman, G. J.; and Hewitt, C. E. "Programming Viewed as an Engineering Activity" MIT A.I. Memo 459. January 1978.

Shoch, J. F. "An Overview of the Programming Language Smalltalk-72". Convention Informatique 1977. Paris, France.

Standish, T. A.; Harriman, D. C.; Kibler, D. F.; and Neighbors, J. M. "The Irvine Program Transformation Catalogue". Department of Information and Computer Science. University of California at Irvine. January 1976.

Steele, G. L. "Debunking the 'Expensive Procedure Call' Myth" MIT Artificial Intelligence Memo 443. October 1977.

Steele, G. L. and Sussman, G. J. "The Revised Report on SCHEME a Dialect of LISP" Artificial Intelligence Memo 452. January 1978.

Strong, H. R. "Translating Recursion Equations into Flow Charts" Journal of Computer and System Sciences. June 1971. pp 254- 285.

Waldinger, R. J. and Levitt, K. N. "Reasoning about Programs" Artificial Intelligence Journal. Vol 5. 1974. pp 235-316.

Ward,S. A. "Functional Domains of Applicative Languages" Project MAC TR-136. September 1974.

Yonezawa, A. "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics" MIT/LCS/TR-191. December, 1977.