MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 119

Three Excerpts from the 1973-74 Project MAC Progress Report

This work was supported by the National Science Foundation under research grant GJ-34671.

March 1975

(Corrected July 1975)

I. Algebraic Specification of Data Types

The realization of an abstract data by a cluster in CLU [8] provides a natural basis for structuring a proof of program correctness. The abstract data type is specified by assertions (axioms) that capture all properties of objects and operations of the data type on which uses of the data type depend. Then a proof of correctness for a program using the cluster consists of two parts: The operations of the cluster are proved to establish the axioms of the data type; and the axioms are used to prove assertions that express correctness of the program.

Since the bodies of cluster operations may utilize further abstract types realized by other clusters that establish corresponding sets of axioms, the structure of a proof of correctness is hierarchical. So long as the problem to be solved by a program can be structured as a hierarchy of abstractions implemented by clusters, the proof of correctness grows linearly with the complexity (depth of the hierarchy of abstractions) of the program.

Steve Zilles has developed an algebraic theory of data type specification that provides a formal basis for constructing sets of axioms for abstract data types and proving correctness of clusters of operations that implement data types [12]. First, the specification of the data type is given in purely algebraic terms and then the derivation of the axioms for the data type is given.

In this work, a data type is viewed as a heterogeneous algebra [2]. A heterogeneous algebra is an algebra defined over more than one set of objects and is given by specifying the family of sets or domains, and the operations defined on them. For the intset data type the domains are the set of objects of the data type, the set integer of integers and the set boolean of truth values. The operator symbols and the functionality of the operations they denote are:

CREATE: → set

INSERT: set X integer → set

REMOVE: set X integer → set

HAS: set X integer → boolean

Each operation of a heterogeneous algebra is defined by associating a function (or "multiplication table") of the correct functionality with its operator symbol. A specification of a heterogeneous algebra must determine the domains over which the algebra is defined and the multiplication tables for the operator symbols. Two heterogeneous algebras will be called similar if they are defined over the same domains and the same set of operator symbols are used in each.

The technique for specification used in this work is to give an algebraic presentation for the desired heterogeneous algebra. A presentation consists of three parts: a set of operator symbols, a set of generator symbols and a set of defining relations. The operator symbols name the operations of the algebra to be defined. The generator symbols denote objects in domains of the algebra.

Using intset as an example, the operators will be CREATE, INSERT, REMOVE, and HAS. The generators will be 0, 1, 2, ... for the integer domain, and true, false for the boolean domain. No generators for the set domain will be needed.

Consider the set of all symbolic expressions formed from the operators and generator symbols, punctuated with parentheses and commas so as to denote compositions of operations in the usual manner. Some examples of expressions, which in algebraic terminology are called words, are

ε_1 = INSERT(INSERT(CREATE, 2), 3)

ε_2 = INSERT(INSERT(CREATE, 3), 2)

ε_3 = REMOVE(INSERT(CREATE, 1), 3)

The set of expressions can be made into a heterogeneous algebra, the word algebra, by defining each operation to yield the expression formed by formally applying the corresponding operator symbol to the expressions that are its operands. This algebra is, in general, too large. Several distinct expressions may denote the same object of the desired data type algebra.

It is the function of the defining relations to indicate which expressions denote the same object and, thereby, express the properties the desired algebra is to satisfy. Each defining relation consists of a pair of symbolic expressions which denote the same object. In the case of intset the pair $(\varepsilon_1, \varepsilon_2)$ is a defining relation because both expressions denote the set {2, 3}.

Although the set of defining relations for interesting abstract data types is generally infinite, it may be finitely represented by relation schemata. A set of relation schemata for the intset data type follows:

1. INSERT(INSERT(s, i), j) ≡ if i = j then INSERT(s, i)
   else INSERT(INSERT(s, j), i)
2. REMOVE(INSERT(s, i), j) ≡ if i = j then REMOVE(s, j)
   else INSERT(REMOVE(s, j), i)
3. REMOVE(CREATE, j) ≡ CREATE
4. HAS(INSERT(s, i), j) ≡ if i = j then true
   else HAS(s, j)
5. HAS(CREATE, j) ≡ false

Schemata 1, 2 and 3 define relations on expressions denoting elements of the set domain. Each relation is generated by substituting appropriate expressions for the set variable S, and the integer variables i and j. Since there are no operators which produce integer results, only integer generator symbols are substituted for i and j. The conditionals on the right hand side mean that alternative expressions are specified according to the outcome of the primitive identity predicate on integer generators i and j. In particular, that e_1 and e_2 are related is shown by schema 1 with i = 2, j = 3 and s = CREATE. Relation schemata 4 and 5 define relations on expressions denoting boolean values.

From a given presentation, the heterogeneous algebra is constructed by defining a set of congruence relations, one for each domain, over the word algebra, and representing the objects of each domain by the congruence classes in that domain. A congruence relation is an equivalence relation in which the results of applying any operation to two tuples of pairwise equivalent operands are equivalent; that is, the equivalence classes are preserved by the operations of the word algebra. The congruence relations used are the smallest congruence relations which contain all the defining relations of the presentation. This means that two expressions are equivalent if and only if there is a sequence of expressions such that the first and last expressions are the expressions in question, and every pair of adjacent expressions in the sequence can be shown to be equivalent using some defining relation.

This construction leads to two theorems on which the utility of algebraic presentations in specifying abstract data types rests.

Theorem 1: A presentation defines an algebra that is unique up to isomorphism, and any equivalence which holds in this algebra can be derived from the defining relations.

Theorem 2: Suppose A is an algebra presented with generators X and defining relations ϕ, and B is an algebra similar to A. Let M be a map $M:X \rightarrow B$ such that every element of B can be obtained from the images of X under M using the operations of B. Then B is a homomorphic image of A if the images of the defining relations ϕ under the map M are all equivalences in B.

Both of these theorems follow from the construction described above. Their primary importance is in proving that two algebras are isomorphic and therefore equivalent. This is the basis for demonstrating the correctness of an implementation of an abstract data type as a CLU cluster.

To use algebraic techniques for showing correctness, it is first necessary to construct an algebra, the concrete algebra, on the representations used in the implementation. This construction is described below: To establish a homomorphic mapping from the abstract algebra onto the concrete one it suffices to show that the defining relations are equivalences when projected into the concrete algebra. If the mapping can be shown to be one-one (injective), it is an isomorphism. Once isomorphism is established, Theorem 1 shows that the defining relations also characterize the concrete algebra and programs using operations of the cluster may be proved correct by using these relations as axioms of the data type.

To this point, the specifications were given in purely algebraic terms. The most commonly used method of proof -- the verification condition method -- is based on having an axiomatization of the data types. The defining relations of a presentation form a partial axiomatization of the data type; they indicate which expressions are equal. The axioms suffice to show any property which depends only on showing the equivalence of expressions. The missing axioms are those which indicate which expressions yield unequal results. These are unnecessary in the

algebraic presentation because the construction of the algebra insures that unless two expressions are equal as a result of the defining relations they will be unequal.

Unfortunately, it is in general no simple matter to construct the axioms which indicate inequalities among the expressions. This is because the algebraic construction is non-effective (in the recursive function sense) and therefore the content of the congruence classes is unknown. In a number of practical cases, however, it is possible to develop an effective procedure for picking a canonical representative of each equivalence class. A canonical representative is any element of the class for which there is an effective procedure for showing its equivalence to all other members of the class. A formula which displays or generates all the canonical representatives is called a canonical form. When such a form exists, it is then possible to construct the axioms for inequivalence. As we shall see below, the existence of a canonic form also simplifies the proof that a homomorphic mapping is one-one and, therefore, defines an isomorphism. For the intset example, the axioms for inequivalence are:

$$(\forall s, i): \quad \mathrm{INSERT}(s, i) \neq \mathrm{CREATE}$$

$$(\forall s, s', i): \quad (s \neq s') \wedge (s \neq \mathrm{INSERT}(s', i)) \wedge (s' \neq \mathrm{INSERT}(s, i)) \text{ implies } \mathrm{INSERT}(s, i) \neq \mathrm{INSERT}(s', i)$$

Very little need be done to define an algebra over the representation used in a CLU cluster. The objects are represented by the chosen representation and the operations are defined by their effects upon that representation. The only problem is the existence of side effects in the operations. This problem can be solved by making the result of operations with side effects by a tuple of values: one for a functional result if any, and one for each operand which is changed by the operation. Symbolic indices or names can be given to the components of the result for purposes of distinguishing them.

Assuming that CLU is given an axiomatic definition [7], the verification condition method of proof can be used to establish input/output assertions for the cluster operations. These are expressed in terms of the representation and become axioms for the operations of the concrete algebra. These axioms are then used to prove that the defining relations are satisfied.

As an example of the procedure for proving the correctness of an implementation, consider the implementation of the intset data type by the cluster in Figure 1. The chosen representation for a value of abstract type is an array of integers in which an integer is an array element if and only if the integer is a member of the abstract set. It will be convenient to formalize this choice by a predicate In(s, i) which tests whether the integer is is represented in the array s:

$$In(s, i) \equiv (\exists j)\ [0 \le j \le HIGH(s)\ \&\ s[j] = i]$$

In this HIGH(s) is the largest index i for which s[i] exists.

Not all integer arrays will be valid intset representations. Specifically, the operations of the intset cluster are coded so that each integer occurs at most once as an element in a representing array. This condition is an invariant of the chosen representation and will be denoted I(s):

$$I(s) \equiv (\forall i, j)\ [0 \le i \le HIGH(s)\ \&\ 0 \le j \le HIGH(s) \Rightarrow i = j \vee s[i] \neq s[j]]$$

The code for the insert operation acts through its side effect on the argument of intset type:

```
insert = oper(s: cvt, i: int);
     if search(s, i) ≤ rep $ high(s) then return;
     rep $ extendh(s, i);
     return;
end insert;
```

```
intset = cluster is create, insert, remove, has, equal, copy;

    rep = array of int;

    create = oper( ) returns cvt;
        r: rep := rep $ create(0);
        return r;
    end create;

    insert = oper(s: cvt, i: int);
        if search(s, i) ≤ rep $ high(s) then return;
        rep $ extendh(s, i);
        return;
    end insert;

    search = oper(s: rep, i:int) returns int;
        for j: int := rep $ low(s) to rep $ high(s) by 1 do
            if i = s[j] then return j;
        return rep $ high(s) + 1;
    end search;

    remove = oper(s: cvt, i:int);
        j: int := search(s, i);
        if j ≤ rep $ high(s) then
            begin
                s[j] := s[rep $ high(s)];
                rep $ retracth(s)
            end;
        return;
    end remove;

    has = oper(s: cvt, i:int) returns boolean;
        if search(s, i) ≤ rep $ high(s)
            then return true
            else return false;
    end has;

end intset
```

Figure 1. The intset cluster.

Using the rule for procedures recently developed by Hoare [7], the proof rule is

$$P^{s}_{INSERT'(s,i)}\ [insert(s,i)]P$$

and the corresponding axiom (input/output assertion) for the behavior of INSERT' is

$$(\forall s, i)\ [I(s) \Rightarrow I(INSERT'(s, i)) \ \&\ In(INSERT'(s, i), i)\ \&\ (\forall j)\ [j \neq i \Rightarrow In(INSERT'(s, i), j) \equiv In(s, j)]]$$

INSERT' is the name of the operation in the concrete algebra. The prime is used to distinguish the abstract and concrete operations which are further distinguished from the cluster operations by capitalization.

Likewise, from the code for the **has operation**

```
has = oper(s: cvt, i: int) returns boolean;
      if search(s, i) ≤ rep $ high(s)
            then return true
            else return false;
      end has;
```

the axiom for the concrete operation HAS' is derived:

$$(\forall s, i)\ [I(s) \Rightarrow HAS'(s, i) \equiv In(s, i)]$$

Axioms for the remaining concrete operations CREATE' and REMOVE' may be similarly derived.

For simplicity, we will assume that the integer and boolean domains of the abstract algebra are isomorphic to the integer and boolean types of CLU and this mapping defines the mapping of the operators into the concrete algebra. Then to establish the isomorphism of the two algebras, it suffices to show that the sets are isomorphic to the set representations. It is not sufficient to view the representation domain as consisting of all arrays that can be generated by composing operations of the cluster because in general many distinct arrays represent the same set. The order in which the integers occur does not matter.

Thus the elements of the rep domain of the concrete algebra must be equivalence classes of integer arrays determined by the equality relation $Eq(s_1, s_2)$:

$$Eq(s_1, s_2) \equiv (\forall i)\ [In(s_1, i) = In(s_2, i)]$$

It is necessary to show that the cluster operations preserve these equivalence classes but once that is done, all that remains to establish a homomorphic mapping is to show the defining relations are satisfied. As an example, we give the proof for schema 4 under the hypothesis $i \neq j$.

Theorem:

$$(\forall s, i, j)\ [i \neq j \Rightarrow HAS'\ (INSERT'(s, i), j) = HAS'(s, j)]$$

Proof:

(1) $i \neq j$ hypothesis

(2) $(\forall j)\ j \neq i \Rightarrow In(s, j) = In(INSERT'(s, i), j)$
from axiom for INSERT'

(3) $In(s, j) = In(INSERT'(s, i), j)$
Modus Ponens (1), (2)

(4) $HAS'(s, j) \equiv In(s, j)$
from axiom for HAS'

(5) $HAS'(INSERT'(s, i), j) \equiv In(INSERT'(s, i), j)$
substitution of INSERT'(s, i) for s in (4)

(6) $HAS'(INSERT'(s, i), j) \equiv HAS'(s, j)$
using the equivalences in (3), (4), and (5)

(7) $i \neq j \Rightarrow HAS'\ (INSERT'(s, i), j) = HAS'(s, j)$
by the deduction theorem on (1) - (6)

Steps 2 and 4 of this proof require that the invariant I(s) of representations hold. That I(s) holds for all elements of the rep domain of the concrete algebra is easily proved by induction using the axioms for the concrete operations since each representation is generated from the empty array by some finite sequence of operations.

The final step in the proof procedure is to show that the homomorphism is an isomorphism. This includes showing that distinct objects in the abstract algebra are mapped to unequal representations in the concrete algebra. This is most easily done when a set of representative expressions -- a set of expressions such that every expression is known to be equivalent to some expression in the set -- is known. Then, if each of these expressions is mapped to a distinct class of the representing array, the set of representative expressions form a canonical set of expressions and the mapping is one-one and an isomorphism. The representative expressions for expressions denoting integer sets have the form

$$\text{INSERT}(\ldots \text{INSERT}(\text{CREATE}, i_1) \ldots, i_n)$$

$$\text{where } i_1 < i_2 < \ldots < i_n$$

That the corresponding representations of the concrete algebra are distinct follows easily from the axiom for INSERT', and the axiom for CREATE' which simply states that the empty array represents the empty set.

With the demonstration of isomorphism, correctness of the intset cluster is established and the relation schemata may be used as axioms of the intset data type for proving assertions about programs using operations of the cluster, without further concern for their implementation.

Besides providing a specification in terms of which the correctness of a cluster can be proven, algebraic specifications have other useful properties. They can be tailored to specify just those properties on which a particular program depends, thereby allowing the greatest possible freedom in the choice of representation. They are suitable input to the implementation generator of an automatic programming system, especially where a canonical form is known. Questions

such as whether two sets of specifications are equivalent can be answered by proving that the algebras defined are isomorphic. And finally, families of algebras can be constructed by adding operations and relations to some fixed set of relations and operations. Algebras seem to provide a basis for a powerful theory of data types.

II. The Binding Model

A programming system supports a community of programmers and permits them to make use of each other's programs. Such a system is said to be modular if programs can be constructed within the system from existing programs without knowledge of the internal operation of those existing programs. For a programming system to be modular, several criteria must be met: there must exist mechanisms which permit any program to be used by any other; each program must have access to all programs and data they need to realize their behavior (programs must be self-sufficient); the behavior of a program must be independent of which program invokes it (programs must be non-discriminatory); and programs must not conflict with one another when used together (any collection of programs must be compatible).

For various reasons, existing programming systems are not modular.

To study how these criteria might be fulfilled in a general purpose programming system, D. Austin Henderson has developed an abstract model -- the Binding Model -- which provides a semantic base for the construction and explication of modular programming systems [6]. The Binding Model has been defined formally using the Vienna Definition Language.

The value of the Binding Model has been established in two ways: Firstly, examples have been used to demonstrate that common constructs of programming languages and operating systems are realizable in the model. Secondly, it has been shown that two kinds of desirable special behavior can be given formal definitions in the model, and that there are large, structurally-defined classes of programs which satisfy these definitions.

For this report, we define one kind of special behavior -- repeatability -- and discuss how our intuitions about repeatability are formalized in terms of the Binding Model.

The Binding Model is a set of states and a state-transition rule which maps each state into a successor state. Each state is composed of two parts: a directed graph of items and a record of the progress of a process in the evaluation of a task. Initial states of the model are those in which no evaluation has taken place; final states are those in which the task has been completed by the process.

Items are nodes of the directed graphs that are states of the model, and are of eight types. Logicals, integers, symbols, and keys are unstructured items: they are nodes of a state from which no edges emanate and are depicted as in Fig. 2a.

Sheaves are items having structure. Sheaves can be regarded as collections of other items in the graph and are depicted as in Fig. 2b, 2c. The items of a sheaf are named by any unstructured items.

Cells are items used to represent variables -- objects subject to change. In any state, each cell is the origin of one arc of the graph (Fig. 2d). The item on which this arc ends is called the cell's contents.

Functionals represent "programs" in the model. A functional has a set of instructions, called its control structure, and a set of associations between identifiers appearing in the control structure and items of the graph. The associations are called bindings, and the set of bindings is called the functional's environment. Figure 2e depicts a functional which computes the sum of two integer items, one of which -- the one bound to the identifier a -- is fixed.

The task of the process in the Binding Model is a functional. The process part of a state records the position of the process in its evaluation of the control structure of the functional. The environment of the functional is used to associate values with the identifiers of the control structure.

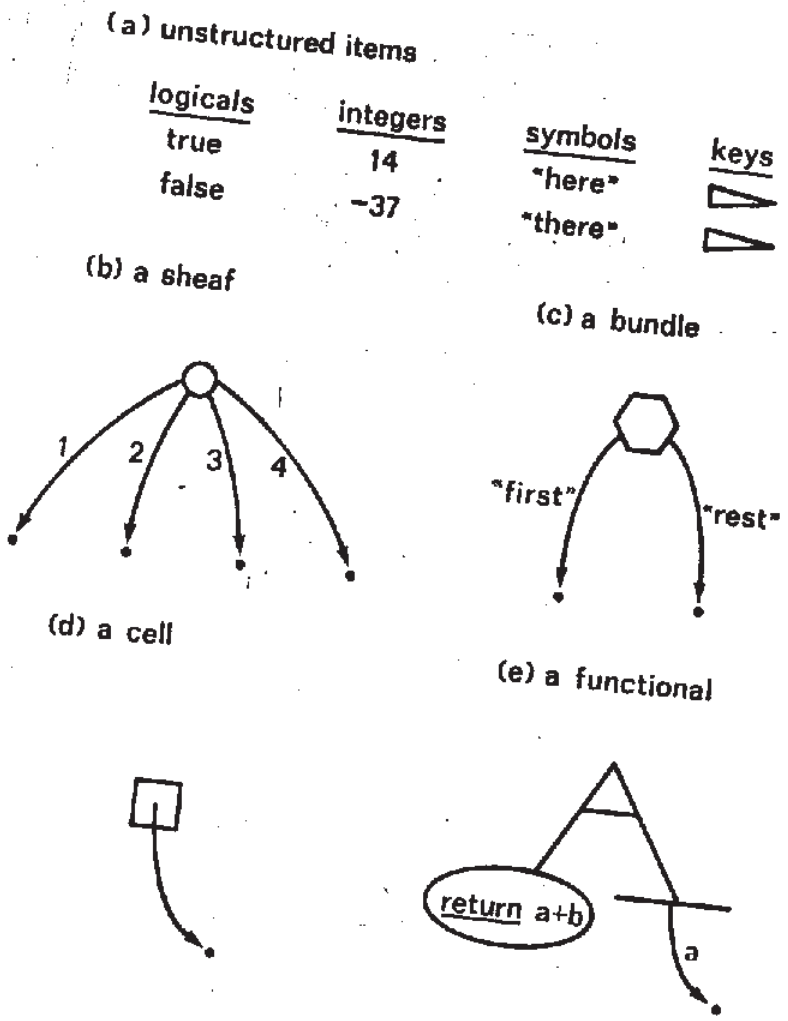

Figure 2. Item types in the Binding Model.

The state transitions of the Binding Model advance the process by performing successive steps in the "evaluation" of the functional which is the task of the process. The basic computation steps are the evaluation of primitive operations which perform such operations as addition of integers, selecting a component of a sheaf, or changing the content of a cell. Figure 3 gives a textual representation for the control structure of a functional. We assume that the environment in which this control structure is evaluated has zero and one bound to the integers 0 and 1, and n bound to some integer. Five forms of control structure are illustrated in this example: A declaration such as

let t5 be add [one, t1] in ⟨CS⟩

means the primitive operation add[one, t1] is to be performed using as operands the values bound to one and t1 in the environment. The result defines a binding for t5 which is appended to the environment, and the control structure ⟨CS⟩ is evaluated using the new environment. The four other forms illustrated express conditional evaluation: if t2 then ⟨CS⟩ else ⟨CS⟩, iteration: repeat ⟨CS⟩, sequential evaluation: begin ⟨CS⟩; ... ; ⟨CS⟩ end, and return of result: return t3.

The primitive operations used in Figure 3 are:

new-cell -- takes a single item and creates a new cell with that item as its contents. The result is the new cell.

contents -- takes a cell; the result is the contents of the cell.

less-than -- takes two integers; the result is the logical item true if the first integer is less than the second, otherwise the result is the logical item false.

add -- takes two integers; the result is the integer which is their sum.

update -- takes a cell and any item and makes the item the new contents of the cell; no result is returned.

In general, application of a functional requires two steps: First, the identifiers of the functional must be bound to items. This is done through repeated use of the bind primitive operation which creates from an existing functional a new one whose environment has one more association. Then the resulting functional is evaluated using the evaluate primitive, which causes a new activation to be set up consisting of the control structure and environment of the functional and a record of "position of evaluation" in the control structure containing the evaluate primitive.

Evaluation of the new control structure proceeds until a return primitive is reached. The return specifies a list of items which become the results of the evaluate primitive in the calling activation, and its evaluation is resumed.

Thus at any time, the position of the process is recorded by a stack of partially-completed activations of functionals. When the stack becomes empty the model is considered to have reached a final state.

Because the model is intended to support the creation of programs, it provides a means of translating representations of control structures into functionals with no associations in their environments. The install primitive takes a control structure representation, and yields a functional having no bindings in its environment.

In a programming system constructed using the Binding Model as a base, program modules would be functionals of the Binding Model. Binding is the means of providing for use of one module by another. Functionals are self-sufficient, and any set of them are compatible. Consequently, a programming system defined in terms of the Binding Model satisfies our criteria for modularity.

The functionals of the Binding Model can exhibit a great variety of behavior. Because several functionals may have bindings to shared structures containing cells, applications of functionals can lead to complex interactions that makes the behavior of a functional difficult to describe or understand. For this reason it is attractive to restrict the possible behaviors of functionals in some way that makes their behavior easier to characterize. A convenient class of functionals are those whose behavior can be described with no mention of time; that is if the functional is applied equivalent inputs on two different occasions, it will produce equivalent outputs. Such a functional is said to be repeatable. Consider, for example, the functional whose control structure is that given in Fig. 3, and whose environment has zero and one bound to $\emptyset$ and 1, respectively. This functional has as input the integer bound to n; its output is an integer which is the sum of the integers up to and including the integer bound to n. This functional will always yield the same output if invoked with the same input. It is therefore repeatable.

A formal definition of repeatability must give precise meaning to "equivalent inputs" and "equivalent outputs" for all items that may occur as inputs and outputs of functionals, including sheaves, cells and functionals. To do this some terminology is necessary.

An item is said to directly enclose any item to which an edge of the graph leads from that item. An item is said to enclose any items which can be reached by following zero or more edges of the graph starting at that time. The closure of an item is the set of items which it encloses. Because the contents of a cell may be changed, these notions are state-dependent when cells are involved.

Difficulties in defining repeatability arise from the possibility that a sheaf may enclose an item that in turn encloses the sheaf -- sheaves (and other structured items) may enclose self-references. In addition, a functional

```
let sum be new-cell (zero) in
let count be new-cell (zero) in
    repeat
        let t1 be contents (count) in
        let t2 be less-than (t1, n) in
        if t2
        then let t3 be contents (sum) in
             let t4 be add (one, t1) in
             let t5 be add (t4, t3) in
                 begin
                     count = t4;
                     sum = t5
                 end
        else let t3 be contents (sum) in
             return t3
```

Figure 3. Computing the sum of the first n integers.

may update cells enclosed by its input items, producing side-effects. Moreover, a functional enclosed by an input item may be evaluated by the functional receiving the input; it would seem that the functional receiving the input should not be considered nonrepeatable if its ill behavior originates from ill behavior of a functional enclosed in its input.

It has proven possible to formulate a definition of repeatability in the Binding Model that agrees with our intuitions about repeatability. The necessary concepts develop as follows:

First we define when two items (possibly in distinct Binding Model states) are similar: Items x and y are similar if there exists a relation between the closure of x and the closure of y under which: (1) items x and y are related, and (2) each pair of related items match, and, if they are structured, the corresponding directly enclosed items are related, and (3) the cells of the closures of x and y are related one-to-one. This relation is said to establish that items x and y are similar. It formalizes the idea that the closures of "equivalent" items must "look alike" and that the sharing patterns for cells must be the same. Figure 4 shows some pairs of similar items.

One might suggest that a functional be called repeatable if similar inputs produce similar outputs. However, this definition would fail to express our intuition when side effects are present: Suppose a functional takes as inputs two cells named a and b, and consider two cases: (1) cells a and b are distinct and have the same integer 1 as their contents; and (2) cells a and b are the same cell (not distinct items) which contains 1. Although the two sets of input items are pairwise similar, the behavior of the functional may be different, for example, if the functional updates one of its inputs.

This problem is remedied by using a generalized notion of similar that considers the closures of all input items and output items jointly. We define

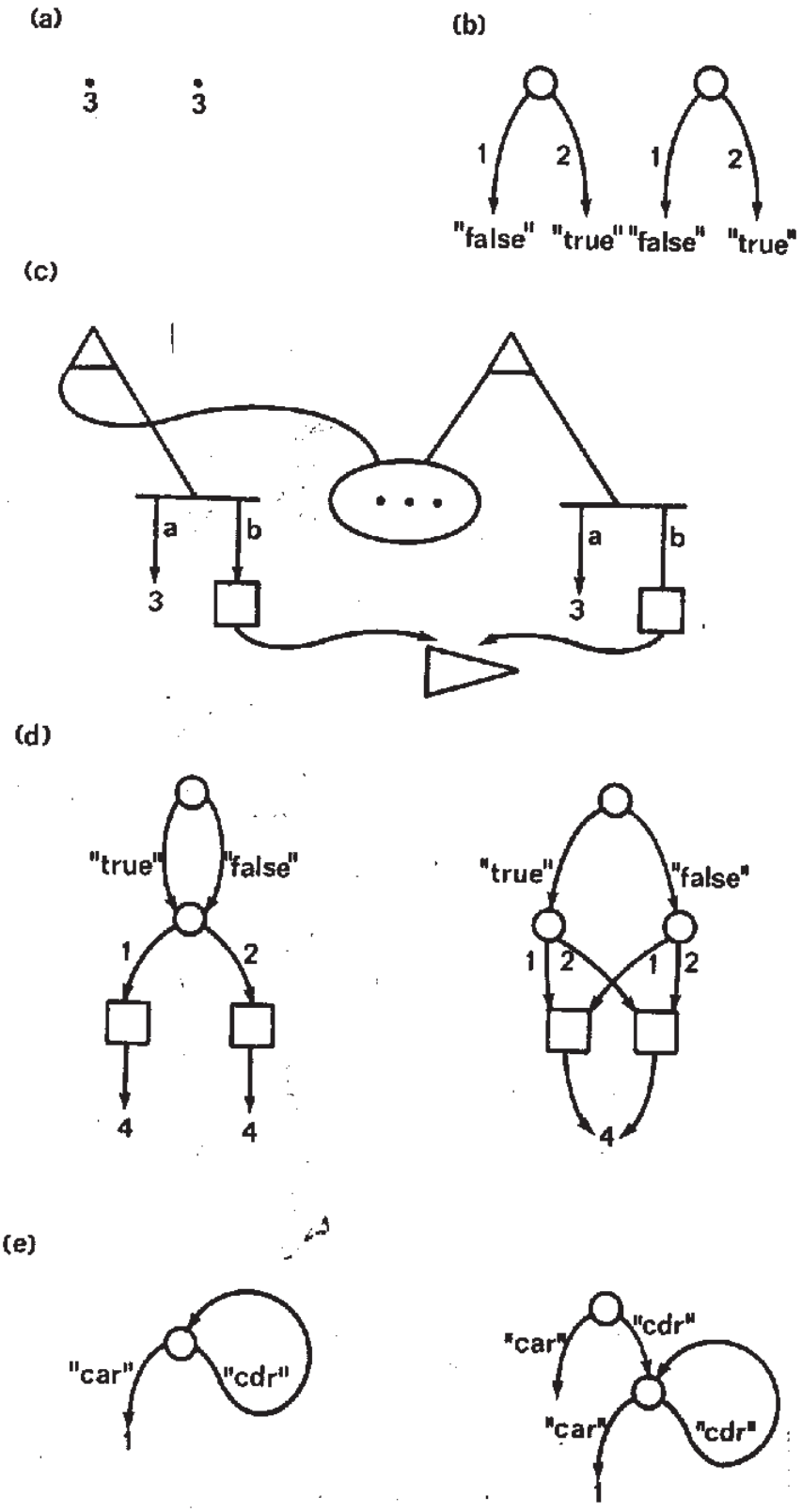

Figure 4. Pairs of similar items.

a similarity to be a relation on sets A and B of items from two (possibly different) states; for each item x in A or B there must be an item y in B or A such that the relation establishes that x and y are similar. There are no unrelated items in the domains of a similarity; and all pairs of related items are established by the similarity to be similar.

A collection of named items is called a package. Inputs and outputs of functionals can be viewed as pakcages by naming the items in the inputs with the identifiers to which they are to be bound, and by naming the items in outputs with the names "result-1," "result-2," and so forth. The notion of similar can be extended to packages by demanding that a single relation exist which establishes the similarity of correspondingly named items of two packages.

Two sets of items (the items of two packages, for example) are said to be covered by a similarity, if the domains of the similarity contain, respectively, the closures of the two sets of items. Let u and v be input packages to a functional f and let F be a similarity that covers u and v. If x and y are output packages resulting from evaluation of f for input packages u and v, and G is a similarity that covers {u, x} and {v, y}, then f is said to extend the similarity of u and v. Thus we might consider a functional to be repeatable if it extends any similarity covering a pair of input packages to a similarity that covers the input and corresponding output packages. But there is one more problem to be solved: We need a way to absolve a functional of responsibility for bad behavior originating from a nonrepeatable functional enclosed by an input item. A natural condition to impose is that a functional have repeatable behavior whenever all functionals enclosed by its input items are repeatable. However, the paradoxes arising from the circularity of this definition (a functional may be enclosed by its own input) are difficult to counter.

We have taken the simpler approach of identifying a simply characterized class of functionals which avoids the circularity. A functional is said to be limited if its control structure has no occurrences of the primitive operators new-key, identical and install. An item is said to be proper if each functional it encloses is limited. The proper functionals can be shown to extend similarities that cover proper inputs and cover the functional itself; under these conditions the results yielded are also proper.

Thus our completed definition of repeatable is: a functional is repeatable if it extends similarities on proper inputs. (Note that the functional to which the definition is applied need not be covered by the similarity.) An immediate consequence of the result stated in the previous paragraph is that proper functionals which enclose no cells are repeatable. For example, the functional constructed from the control structure represented in Fig. 3 by binding zero and one to the integer items Ø and 1 is proper and cell-free. Consequently it is repeatable.

Another class of specially-behaved functionals are those which are repeatable when viewed as functions from sequences of inputs to sequences of outputs. The doctoral thesis [6] includes discussion of the intuitions and formalization of this notion of "sequence-repeatability."

III. Data Flow Computer Architecture.

The concept of data flow opens up attractive new approaches to the architecture of stored program computers. In last year's report [9] we described an unusual structure for a processor for the stream-oriented computations that characterize a number of important signal processing applications such as waveform generation, modulation and filtering. Recently, we have discovered how the principles embodied in this Elementary Data Flow Processor can be generalized and applied in a Basic Data Flow Processor that implements conditional and iteration constructs and incorporates a two-level physical memory hierarchy. Further development of these architectural concepts to achieve an exact realization of a complete data flow language appears possible, and is the principal goal of our continuing research program in computer architecture.

The Elementary Processor has the structure shown in Fig. 5. There are four major sections:

Instruction Cells

Arbitration Network

Operation Units

Distribution Network

The sections communicate by packet transmission using only the channels shown in the figure. Each instruction corresponds to an operator of the data flow program, and resides in an Instruction Cell awaiting arrival of its operands. An instruction together with its operands form an instruction packet that flows through the Arbitration Network and is directed to an appropriate Operation Unit according to the instruction's operation code. The result values produced by Operation Units are paired with destination addresses from the instruction to form result packets that are sent through the Distribution Network to deliver

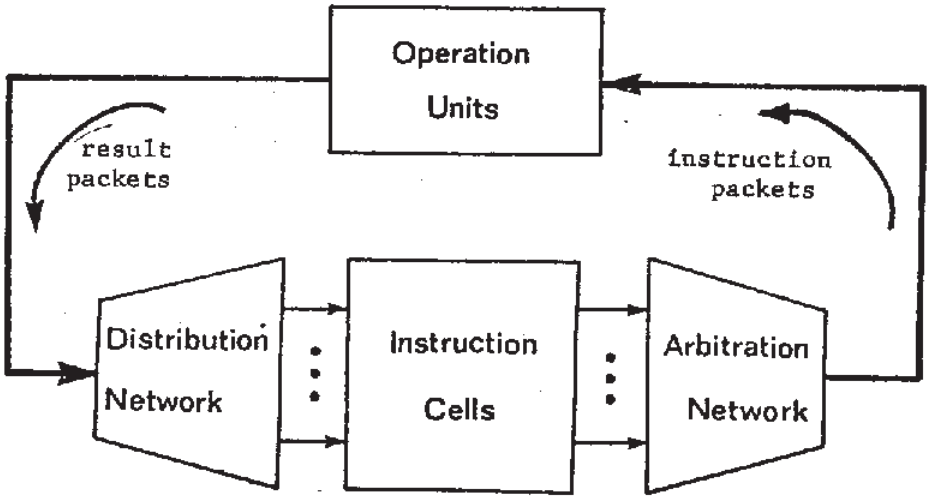

Figure 5. The elementary data flow processor.

operand values to other instructions. Interestingly, the address fields of instructions tell the processor where to deliver the results of an operation instead of where to fetch operands. The Arbitration and Distribution Networks behave like sorting networks -- the former sorts instruction packets by operation code; the latter sorts result packets by destination address.

Detailed logic designs have been developed for each section of the Elementary Processor [4], and its performance and applicability are being analyzed to determine the desirability of constructing a prototype machine.

The Basic Data Flow Processor extends the language of the Elementary Processor to include deciders, boolean operators, gates and merge nodes as well as operators and links. The level of data flow language is precisely the formal data flow schemas used in our theoretical studies, and corresponds in expressive power to the formal while programs of Ashcroft and Manna [1].

The structure of the Basic Processor is shown in Fig. 6. To handle decisions whose results are truth values, the Elementary Processor is augmented with Decision Units. The Decision Units produce control packets consisting of a truth value and a destination address which flow through the control network to reach the Instruction Cell that contains the destination operand register.

At the Instruction Cell, the truth value is either a boolean operand if the Cell represents a boolean operator, or the truth value instructs the Cell to accept or reject the next arriving data value. In this way the gating of operand values is incorporated into the Instruction Cells of the processor

In contrast with the Elementary Processor in which all instructions are equally active, activity of a basic data flow program shifts among program parts according to the outcomes of decisions. Thus it is no longer efficient to hold all instructions of the data flow program in the Instruction Cells of the Processor. Accordingly, an Instruction Memory is included in the machine and the Instruction Cells are organized to act as a cache, retaining the most active

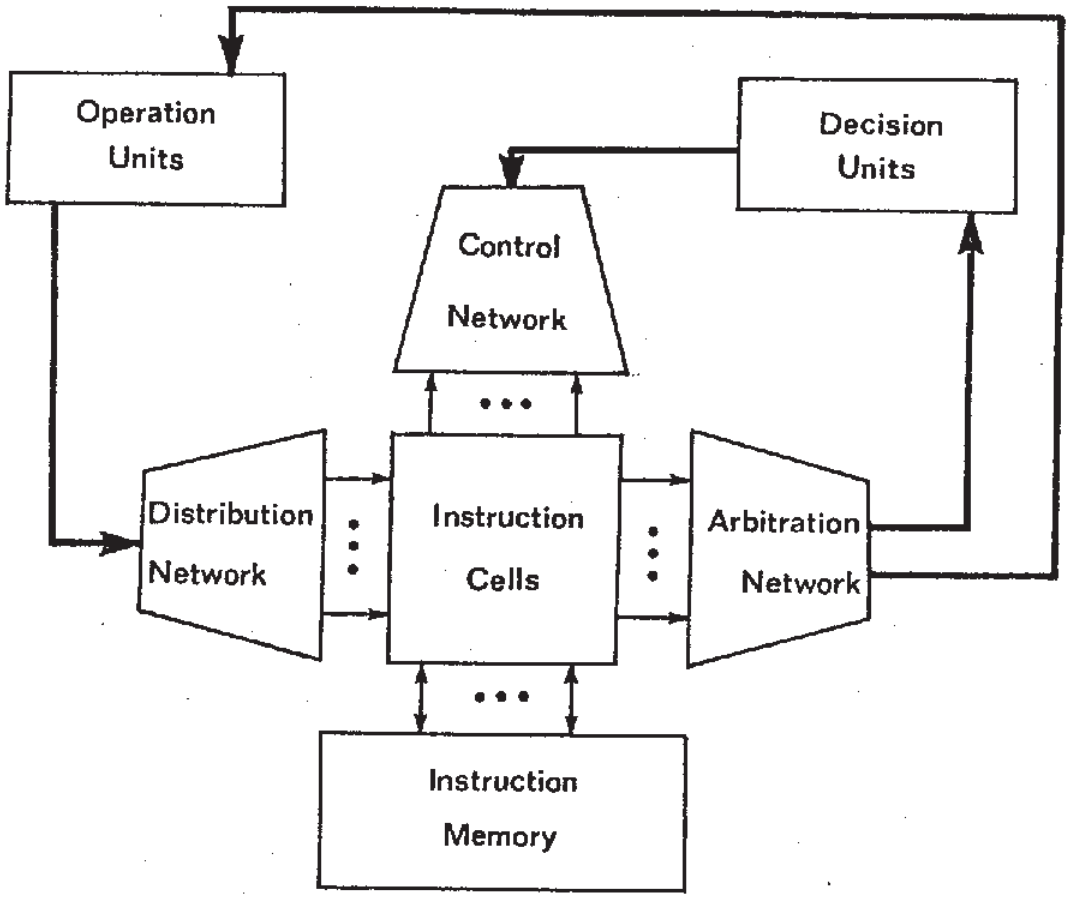

Figure 6. Basic data flow processor with two-level memory.

instructions of the data flow program.

The functional structure of the Basic Data-Flow Processor is described in a paper by Dennis and Misunas [5]. The Basic Processor demonstrates how the Elementary Processor can be augmented with a decision capability and a multi-level memory capability and is a significant step in generalizing the concepts to a complete data flow language.

In doctoral research, James Rumbaugh [11] has explored a more direct approach to the design of a highly parallel machine for programs expressed in a data flow representation equal in expressive power to the data flow procedure language discussed in Dennis [3].. The machine has the structure shown in Fig. 7a. The principal working elements of the machine are the Activation Processors, each of which is at any time given exclusive responsibility for the progress of one activation of a data flow procedure. Since procedure activations in these data flow programs have no side effects, distinct activations can always be executed simultaneously by separate activation processors without need for synchronization. The Program Memory holds the coded text of each procedure of a data flow program. The coded text defines the initial contents of the local memory when an activation processor is assigned by the Scheduler to a newly initiated procedure activation. The Swap Memory holds Local Memory images of activations that have ceased activity (are dormant) and are awaiting completion of other activations invoked by apply instructions.

The Scheduler handles the allocation of Activation Processors to procedure activations and the passing of argument and result values between activations. Assignment of an available processor is required when an Activation Processor executes an apply instruction in its Cell/Ref Functional Unit, or an activation terminates and the activation from which it was invoked is dormant. An Activation Processor is released for reassignment by the Scheduler only when the

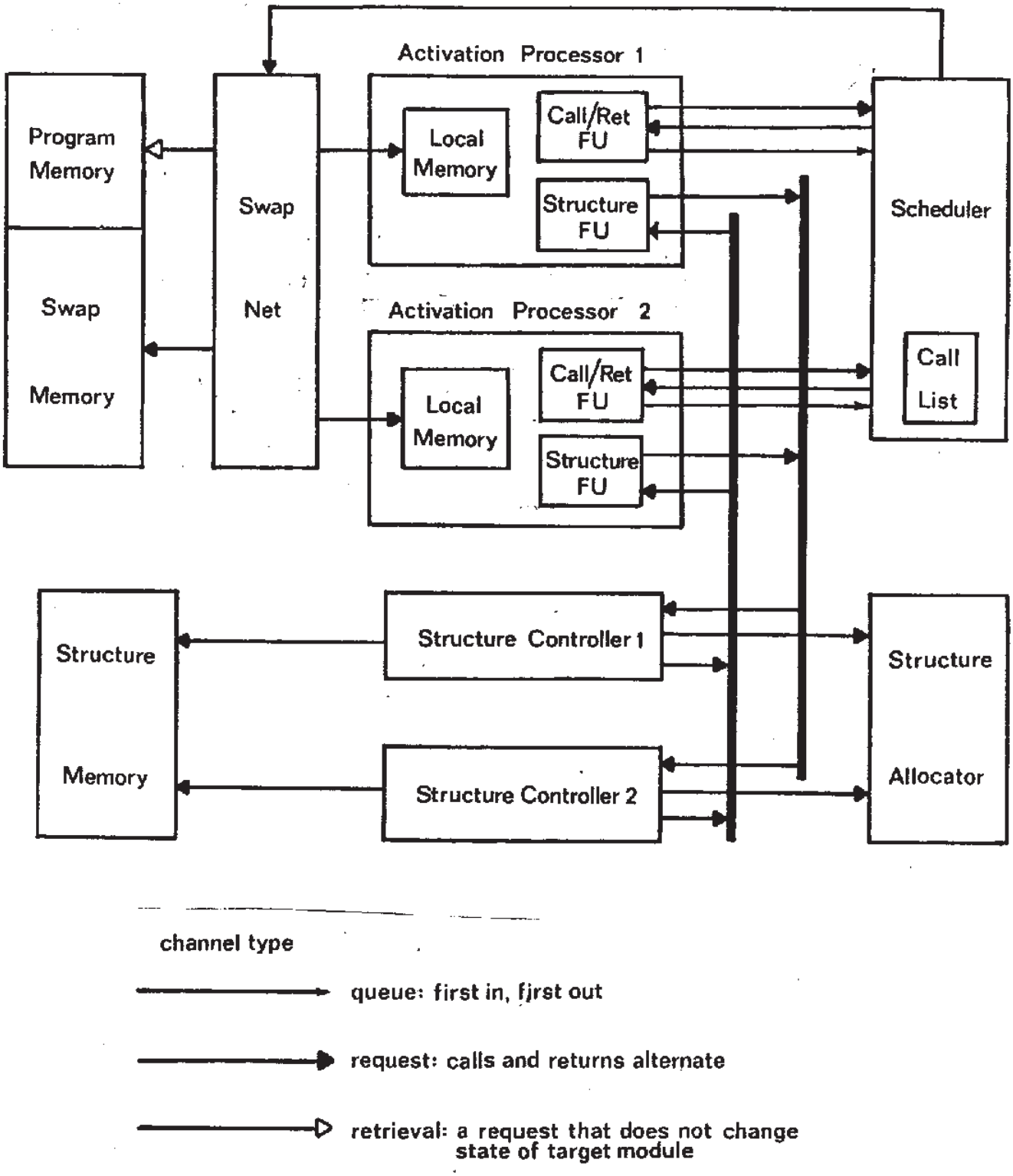

Figure 7. Structure of a data flow machine.

procedure activation terminates (and its result value is returned to the calling activation through the Scheduler), and when the activation becomes dormant waiting for a return value. Since the number of invocations generated may exceed the number of activation processors, the Scheduler contains a Call List of potential activations waiting for allocation of processors.

Data structures in the language of this machine are trees whose nodes denote substructures, and such that the arcs emanating from any node are labelled by a sequence of integers starting with 1. This is the same class of structured data objects used in the Symbol machine [10]. The basic operations on data structures are versions of the select and append operations discussed in Dennis [3]. The select operation obtains the component of a structure indexed by a given integer; the append operation produces a new structured value obtained by substituting a given value for a specified component of a given data structure.

The data structures accessed during a procedure activation may be arbitrarily large, and cannot generally be held in the Local Memory of an Activation Processor. Moreover, a large data structure may be accessible to several concurrent activations and it is best not to make unnecessary copies. For these reasons, data structures are held in a Structure Memory and accessed through requests processed by Structure Controller units. Each data structure node is represented in a segment in Structure Memory identified by a unique address; this address is the representation of a structured value in the Activation Processors. The Structure Allocator unit controls the allocation of Structure Memory locations to data structure nodes. Since the allowed operations on data structures do not permit existing data structures to be altered, formation of cyclic data structures is impossible, and the reference count mechanism can be

used by the Structure Allocator to determine when segments of Structure Memory become free.

The structure of an Activation Processor is shown in Fig. 8. The Local Memory consists of three parts: the Instruction Memory that holds the instructions of the data flow procedure assigned to the processor; the Operand Memory that holds values carried by the conceptual tokens on the arcs of the data flow procedure; and the Enable Memory that holds an enable count for each instruction. The enable count is simply the number of remaining operand values required before the instruction is enabled for execution. When the number becomes 0 the instruction address is entered in the Activity Queue.

The Activity Queue, Decoder, Dispatcher, the several Functional Units (FU), and the Updater form a circular pipeline system that can process many instructions concurrently. The execution of each instruction involves one traverse of the circuit. The Decoder takes an address from the Activity Queue and forms an instruction packet by fetching the instruction from Instruction Memory and the operand values called for by the instruction from Operand Memory. The Dispatcher delivers the instruction packet to the appropriate Functional Unit according to the opcode of the instruction. The Updater receives each result value of instruction execution together with the address of the Operand Memory location which is the destination of the value, and the address of the instruction whose enable count should be decremented. The Updater performs these actions and notes whether the successor instruction has become enabled (all operands present -- enable count zero). If so, the instruction address is entered in the Activity Queue.

The number of concurrent activities in an Activation Processor is initially 1, increases with execution of a wye instruction which produces two result values that are copies of its single operand, and decreases when two results become operands for one instruction, for example, the add instruction. The Activity

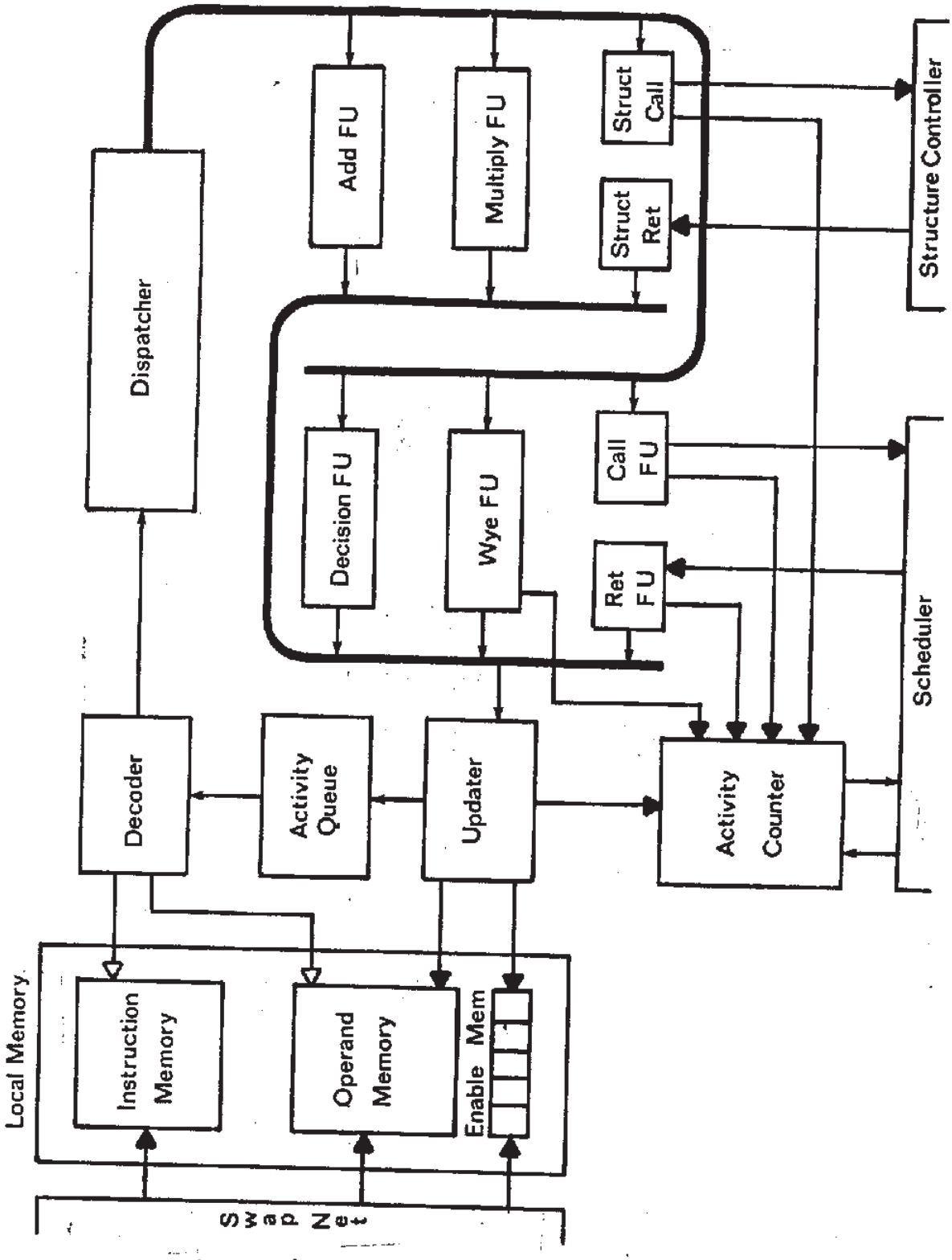

Figure 8. Structure of an activation processor.

Counter maintains this count of activities on the basis of signals from the Updater, the Wye Functional Unit and the Structure Functional Unit which handles wye instructions with structured operand values. An apply instruction (which calls for initiation of a new procedure activation) is processed by the Call/Ret Functional Unit. Since the new activation may run a long time before terminating, the activity count is decremented at initiation and incremented at termination.

An activity count of zero means that either the activation has terminated, or the activation is dormant pending termination of an inferior activation.

In either case the Scheduler is notified that the Activation Processor is available for reallocation. In the case of termination, the Scheduler reallocates the calling activation to an available Activation Processor if it is dormant, and passes the returned value to its Call/Ret Unit.

All instructions that access or operate on structured values are passed to a Structure Controller for execution. This is done so that correct reference counts can be maintained. In particular, a wye instruction for a structured value must increment the associated reference count.

References

1. Ashcroft, E., and Z. Manna. The translation of 'go to' programs to 'while' programs. Information Processing 71, North-Holland Publishing Co., Amsterdam 1972, 250-255.

2. Birkoff, G., and J. D. Lipson. Heterogeneous algebras. J. of Combinatorial Theory 8 (1970), 115-133.

3. Dennis, J. B. First Version of a Data Flow Procedure Language. Computation Structures Group Memo 93-1, Project MAC, M.I.T., August 1974.

4. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974, 402-409.

5. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975, 126-132.

6. Henderson, D. A., Jr. The Binding Model: A Semantic Base for Modular Programming Systems. Project MAC Technical Report, M.I.T., forthcoming.

7. Hoare, C. A. R., Procedures and parameters: an axiomatic approach. Symposium on Semantics of Algorithmic Languages (E. Engeler, Ed.), Lecture Notes in Mathematics 188, Springer-Verlag, 1971, 102-116.

8. Liskov, B. A Note on CLU. Computation Structures Group Memo 112, Project MAC, M.I.T., November 1974.

9. Progress Report 1972-1973, PR3X, Project MAC, M.I.T., 67-77.

10. Richards, H., Jr., and C. Wright, Jr. Introduction to the SYMBOL-2R programming language. Proceedings of a Symposium on High-Level-Language Computer Architecture, SIGPLAN Notices 8, 11 (November 1973), 27-33

11. Rumbaugh, J. E. A Parallel Distributed Machine Architecture for a Data Flow Base Language. Ph.D Thesis, Department of Electrical Engineering, M.I.T., in preparation.

12. Zilles, S. N. Data Algebra: A Specification Technique for Data Structures. Ph.D Thesis, Department of Electrical Engineering, M.I.T., in preparation.