An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages

by

John T. O'Donnell

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 180

An Architecture that
Efficiently Updates Associative Aggregates
in Applicative Programming Languages

by

John T. O'Donnell

September, 1985

To appear in *The Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture* in Nancy, France, September 16–19, 1985.

An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages

John T. O'Donnell

Computer Science Department
101 Lindley Hall
Indiana University
Bloomington, Indiana 47405

Abstract

Applicative (also called functional) programming systems prohibit side effects, including assignments to variables. This restriction has several advantages, including referential transparency and potential parallel program execution. A major disadvantage, however, is that aggregate data structures become very expensive to maintain: when the programmer updates a single element in an aggregate, many applicative language implementations must completely recopy the aggregate. This paper solves the aggregate update problem with a two-level architecture: a microprogram maintains "associative aggregate" data structures, and a hardware memory design (the Associative Aggregate Machine) implements powerful insertion, deletion and searching operations required by the microprogram. The Associative Aggregate Machine contains a linear sequence of cells comprising storage and combinational logic. Each cell is connected to its predecessor and successor, so the sequence of cells forms a shift register that supports insertion and deletion. In addition, a binary tree of combinational logic nodes performs fast associative searching through the sequence of cells. The Associative Aggregate Machine architecture is extremely regular and is well suited for VLSI implementation.

1. Introduction

Purely applicative languages (also called functional languages) have many advantages, but their implementation raises several problems. This paper introduces a new solution to one of them, the "aggregate update problem". We define an associative aggregate data structure that satisfies the representation and accessing requirements of lists, vectors and environments, and we show how to implement the aggregate access functions very efficiently (all aggregate accesses require a constant number of storage cycles). Our solution does not rely on compile-time analysis of the applicative program. Instead, it uses a new computer storage architecture that allows program interpretation and that always works.

An aggregate is a data structure containing many elements that are individually accessible. Common examples of aggregate structures are lists, streams and vectors. Two operations are necessary to manipulate aggregates: **lookup**, which fetches a value from the aggregate; and **update**, which stores a new value into the aggregate (possibly replacing a previous value).

The **update** operation should return a new aggregate and leave all others (including the one being updated) unaffected. Imperative languages do not insist on this conceptual view of **update** because there is always only one extant version of an aggregate: the one that resulted from the most recent **update**. Thus implementation of the **lookup** and **update** operations is trivial for imperative languages, because they compile into **fetch** and **store** instructions respectively. In contrast, applicative languages prohibit side effects, so **update** must return a new aggregate, leaving the old one unaffected. Simply using **store** for the **update** would change the original aggregate. The usual way to implement the applicative **update** is to recopy the original aggregate, making it extremely expensive compared with an imperative **update**.

One way around the applicative aggregate update problem is to try to arrange the program so as to make the trivial imperative **update** safe. Using this approach, Hudak and Bloss [5] give several techniques based on applicative source program analysis that often allow safe implementation of **update** with a **store**. This happens when the original aggregate is inaccessible, so side effects to it are harmless. Such techniques are extremely successful in many applications. In particular, imperative numerical

analysis programs typically consist of sequences of **lookup** and **update** operations, where **lookup** will be applied only to the aggregate resulting from the most recent **update**. Most such **update** operations may be replaced safely by **store** operations, since the original aggregate will never be used again. Related discussions of program analysis are in [15] and [11].

One disadvantage of relying on program analysis to find safe ways to perform destructive updates is that compilation (or at least program analysis before execution) becomes necessary. Several techniques for constructing applicative programming environments are better suited to interpretation, and we do not want to be forced into compilation just in order to gain acceptable efficiency. Another disadvantage is that program analysis techniques do not always work: **update** still may require recopying. Programmers might develop styles that enable the compiler to handle all updates efficiently, but it is undesirable to constrain programming styles unnecessarily.

A better solution to the aggregate update problem would ensure that all applications of **update** and **lookup** execute in constant time. That would allow programmers to express their algorithms as they wish, and would provide better support for advances in programming style. This paper develops such a solution. We began by observing that the **fetch** and **store** instructions of conventional architectures fail to implement **lookup** and **update** efficiently for applicative languages. But our conclusion is not that something is wrong with **update** or **lookup**, or with applicative languages; instead, something is wrong with **fetch** and **store**. Our approach is to invent a novel architecture with powerful instructions that do make applicative aggregate access efficient.

The remainder of the paper describes an architectural solution to the aggregate update problem based on the Associative Aggregate Machine (AAM). Section 2 defines associative aggregates and their access functions, and Section 3 shows how the AAM architecture represents them. Section 4 gives the AAM's instruction set, which is much more powerful than the **fetch** and **store** of conventional architectures. Section 5 shows how to manipulate the representations of associative aggregates using that instruction set, and Section 6 then shows how to implement the AAM architecture directly in hardware. Section 7 concludes by discussing the nature of parallelism in the AAM.

2. Associative Aggregates

There are several ways to access information in aggregates. List aggregates have **car** (or **head**, **first**, *etc.*) and **cdr** (or **tail**, **rest**, *etc.*) operations. Vector aggregates allow access to elements through their indices. Lists and vectors appear to have very different properties, but we have shown [13] that a computer storage architecture can implement a combined list/vector aggregate data structure, in which *all* the usual list and vector operations execute in a constant number of storage cycles (from one to six). For example, it is possible to insert an element into a list/vector aggregate, delete another element, and then index into the middle of the resulting aggregate in constant time. The result of that work was to unify the notions of "list" and "vector".

This paper deals with associative aggregates. An *associative aggregate* is a set of *bindings*, where each binding is an ordered pair *(name, value)*. The **lookup** and **update** functions both refer to values through their associated names. An aggregate may contain only one binding for a name (the data structures used to *represent* aggregates may contain several bindings for a name, but only the most recently bound value is used; see Section 3). Associative aggregates can represent lists and vectors by using the index of an element as its name at the expense of doubling the size of the representation. In a general applicative machine it would be better to combine the techniques of [13] with the techniques introduced in this paper in order to eliminate that expense. Associative aggregates can also directly represent environments, so the applicative language interpreter can use **lookup** and **update** for efficient lambda binding and variable evaluation. We discuss our implementation in terms of "associative aggregates" rather than "environments" because it is useful for many applications in addition to environment manipulation.

Two functions are available to the applicative programmer for manipulating associative aggregates:

$$\texttt{update}(x, n, t, v) \quad \text{and}$$

$$\texttt{lookup}(x, n),$$

where x is a reference to an aggregate, n is a name, and v is a value of type t to be bound to n. A reference x to an aggregate must be either **nil** (the empty aggregate),

or the result of an application of **update**. Both the **update** and **lookup** functions leave all existing values undisturbed. The **lookup** function returns the value bound to n in x, and the **update** function returns a new aggregate equivalent to x except that n is bound to v with type t. Thus

$$\texttt{lookup}(\texttt{update}(x, n, t, v), m) \quad \equiv \quad (m = n \longrightarrow v, \quad \texttt{lookup}(x, m)).$$

Note that the **update** and **lookup** functions provide a natural implementation of environments [4]. There is no function for explicitly creating aggregates; the programmer must instead build up large aggregates by applying **update** to a sequence of aggregates beginning with **nil**.

In addition to **update** and **lookup**, there is a function **destroy** available only to the storage manager (but not to the applicative programmer). When the storage manager knows that there are no live references to aggregate x, it executes **destroy**(x) to delete all parts of the representation of x that are not shared by the representation of any other aggregate. Most garbage can be found associatively; this is similar to reference count reclamation. If circular data structures are present garbage collection will also be necessary. With minor extensions, the AAM can support an extremely fast garbage collector.

Descriptions of language interpreters often take the form of an algorithm manipulating data structures, with the implicit assumption that the algorithm is executing on a processor that issues **store** and **fetch** requests to a conventional storage with addressable words. In contrast, the implementation described in this paper involves two levels of abstraction:

1. *The language interpreter*, which executes on a processor that issues instructions to an active storage architecture. By exploiting the powerful features of the active storage, the interpreter algorithm is able to manipulate associative aggregates; other useful data structures supported by the active storage are described in [12, 13].

2. *The active storage architecture*, called the *Associative Aggregate Machine* (AAM). The AAM architecture must be implemented directly in hardware. An emulator

running on a conventional microprogrammed host cannot implement the AAM efficiently.

Therefore our solution to the aggregate update problem requires a new data structure, algorithms to manipulate it, and hardware that can execute the instructions issued by those algorithms. Thus

$$\textit{algorithms} + \textit{data structures} + \textit{architecture} = \textit{implementation}.$$

3. Representation of Associative Aggregates

The representation of associative aggregates must support two operations: fast **lookup** and fast **update**. It is straightforward to find a representation that satisfies either operation alone. For example, with a conventional linked list representation, **update** requires a constant number of storage cycles, but searching the linked list for a binding requires iteration, making **lookup** expensive. Similarly, conventional address indexing can implement **lookup** in a constant number of cycles, assuming a fixed vector representation for the aggregate, but updating an element inside a vector would either require recopying the vector (making **update** expensive in both space and time) or maintaining a file of transactions or exceptions (making **lookup** expensive). Thus the problem is to find a representation that supports both **lookup** and **update**. The AAM uses a vector representation with special hardware support for insertion and lookup.

The AAM storage consists of a linear sequence of cells. The content of a cell is called a *word*. The interpreter frequently shifts words through the storage, so they are not accessible through the addresses of the cells that hold them. Consequently words are accessible through their relative positions or contents, rather than through fixed storage addresses. We refer to the two endpoint cells as the leftmost and rightmost, respectively, and they are connected to the Left Port and Right Port.

Each cell contains two fields (*type* and *value*) and three flags (*select*, *mark* and *attached*). Figure 1 shows the notation for displaying the contents of a cell. If the *select*, *mark* or *attached* flag in a cell is **true**, then the corresponding symbol (•, ○, ▷) appears in the box notation.

Each cell responds independently to every instruction, and its action usually depends on the settings of its flags. The *select* flag implements associative pointers, and

select	•	indicates active cell
mark	○	marks a span of adjacent cells
attached	▷	indicates continuation of data
type		data type of value
value		data value

Figure 1. Storage Cell Fields

the *mark* flag marks a subset of cells that will respond to certain instructions. The *attached* flag plays a crucial role in representing aggregates. The representations consist of one or more *spans*. A span is a sequence of adjacent words comprising a single unit of information. The last cell in a span has *attached* = **false**, and all the others have *attached* = **true**.

The *type* field indicates the data type of the value stored in the *value* field (thus the AAM is a tagged memory [7]). The choice of data types depends on the intended application for the AAM; we assume that at least five types exist: *def*, *ref*, *cnt*, *name* and *int*. Any cell containing *type*=*ref* and *value*=V represents a pointer to the cell with *type*=*def* and *value*=V. Thus a *ref* is a pointer and a *def* is the target of a pointer.

Figure 2 illustrates aggregate representation by showing the result of a sequence of **update** operations. Each execution of **update** requires several microinstructions (and all microinstructions require exactly one AAM storage cycle). Only the final results of the **update** instructions appear in Figure 2; Section 5 describes how the microinstructions issue AAM operations in order to produce the final result. It is important to note that there is a constant bound on the number of microinstructions needed to implement **update** (seven cycles) as well as **lookup** (five cycles). The figure does not show the value of the *select* and *mark* flags, since the microprogram needs

them only during execution of an individual **update** or **lookup**, and their settings are irrelevant to the representation of aggregates.

The microprogram represents a binding (n, v), where value v of type t_v is bound to name n, with a pair of words in adjacent cells. The leftmost word has *type* = *name* and *value* = n, and its *attached* flag is **true**, indicating that its successor is part of the same data structure. The successor contains the binding: *type* = t_v and *value* = v. The *attached* flag of the word containing the binding value may be either **true** or **false**, depending on the presence of other bindings in the same aggregate.

Following a pointer to an aggregate requires associative searching. Since words may be shifted from one cell to another, pointers are not representable as storage addresses. However, the microprogram cannot access the binding (n, v) simply by searching for a cell containing [*name* n], because an unrelated aggregate could have its own binding for n. Therefore the microprogram maintains an explicit reference to every aggregate through a word with *type* = *def*, *value* = p and *attached* = **true**, where p is a unique identification number. Whenever the microprogram creates an aggregate, it finds an unused identification number to serve as the *value* of the *def* word. The AAM is always able to generate a unique identification number in one cycle; [12] explains the method used.

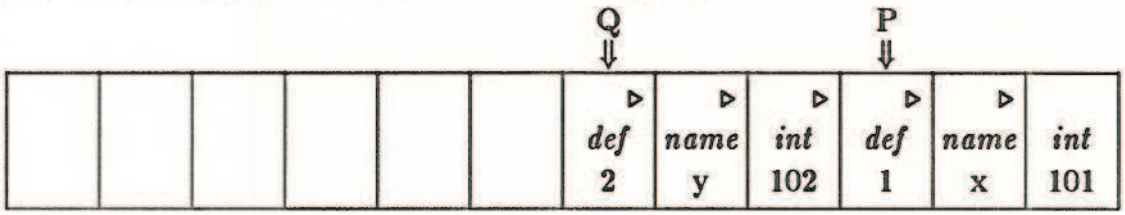
Figure 2a shows the result of **update**(**nil**, x, *int*, 101), which creates a new aggregate by updating **nil**. The **update** function generated identification number 1 and returned [*ref* 1], a reference to the aggregate P.

Figure 2b shows the representation after creation of Q = **update**(P, y, *int*, 102). The reference to P is unchanged, and the *attached* flags indicate that the span beginning with [*def* 2] continues through the binding for P, so the following bindings exist: $P \equiv \{(x\ 101)\}$ and $Q \equiv \{(y\ 102)\ (x\ 101)\}$.

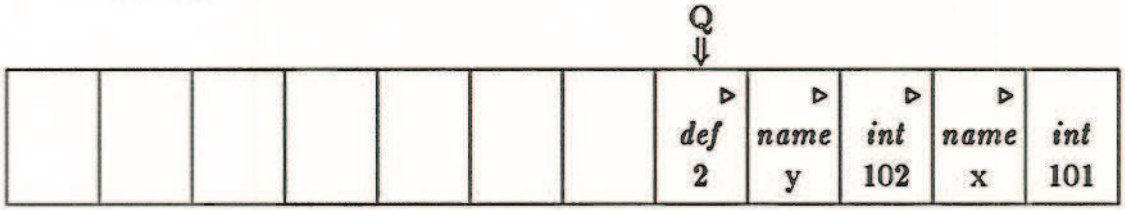
When the microprogram destroys the reference to P (Figure 2c) it can reclaim only the word containing [*def* 1] because the binding for x is shared with aggregate Q. The result is a compact representation of Q. Programs that keep live references only to the result of the most recent update can thus maintain compact representations of their current aggregates, and irrelevant *def* words will not proliferate. In the example, the microprogram knows that the binding for x must be retained because the *attached* flag

(a) $P :=$ **update**(**nil**, x, *int*, 101); *result:* $P \equiv$ [*ref* 1]

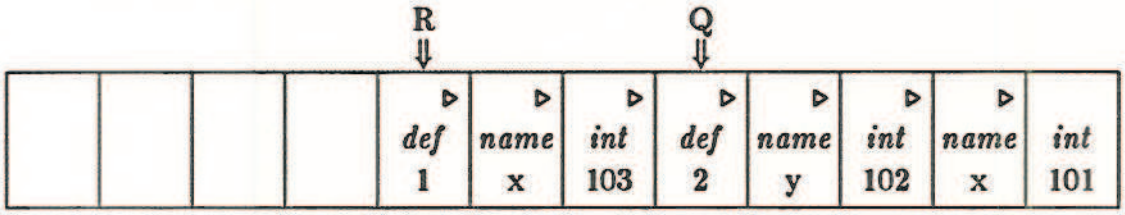
(b) $Q :=$ **update**(P, y, *int*, 102); *result:* $Q \equiv$ [*ref* 2]



(c) **destroy**(P); *result: release id 1*



(d) $R :=$ **update**(Q, x, *int*, 103); *result:* $R \equiv$ [*ref* 1]



(e) $S :=$ **update**(Q, z, *int*, 104); *result:* $S \equiv$ [*ref* 3]

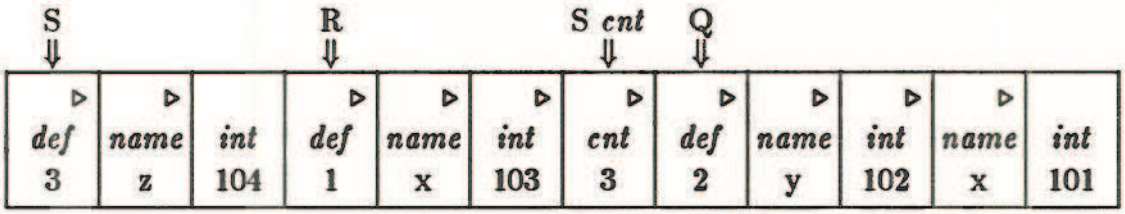


Figure 2. Examples of Associative Aggregate Representation.

of the predecessor word is **true**, indicating that the binding of x is being shared with another aggregate. The **destroy** microcode checks the *attached* flag of the predecessor of the aggregate being destroyed, and deletes the entire aggregate if the flag is **false**, indicating that the representation is unshared.

An associative aggregate may have only one binding for each name, but its representation may contain several. For example, in Figure 2d, R is an **update** of Q so its representation shares the representation of Q. Consequently the representation of R contains three bindings, two of them for x: $R \equiv \{(x\ 103)\ (y\ 102)\ (x\ 101)\}$. When **lookup** searches R for the binding of x, it will choose the correct one by looking only at the leftmost binding representation. Since **update** always inserts new bindings to the left of existing aggregates, the more recent binding to a name will be to the left of an older binding. Note that the identification number for P (1) was reclaimed when P was destroyed, and that number now identifies R. The system must allocate and reclaim identification numbers just as it allocates and reclaims storage words.

A more complex representation results when two distinct aggregates are created by applications of **update** to the same older aggregate. Figure 2e illustrates the result of defining S as an **update** to Q, while R (an earlier **update** to Q) still exists. The representation of S must reside in the AAM storage to the left of the representation of R, because R is older. However, S must not share the representation of R since it contains the binding $(x\ 103)$. Therefore the microprogram clears the *attached* flag in the last word of the new binding (*i.e.*, [*int* 104]), indicating that the first span representing S does not extend into the span for R. The microprogram also puts a second span into the representation of S by inserting the word [*cnt* 3] just to the left of the *def* for Q (3 is the identification number assigned to S). The **lookup** operation will search all spans belonging to the aggregate, whether they begin with a *def* or a *cnt* word (see Section 5). To summarize, Figure 2e shows the following represented aggregates:

$$Q \equiv \{(y\ 102)\ (x\ 101)\}$$

$$R \equiv \{(x\ 103)\ (y\ 102)\ (x\ 101)\}$$

$$S \equiv \{(z\ 104)\ (y\ 102)\ (x\ 101)\}$$

4. AAM Instruction Set

A conventional computer storage can execute two instructions, **fetch** and **store**, which are sufficient for the implementation of ordinary data structures. In contrast, the AAM is an active storage unit that can execute nine instructions useful for implementing associative aggregates. In addition to store and fetch, the AAM instructions can perform operations that would require iteration using a conventional storage. This section describes the AAM's instruction set. Section 5 shows how the **update**, **lookup** and **destroy** functions are implemented with these instructions, and Section 6 describes how to implement them in hardware.

1. **select**(x)
   Each cell with *value*=x and either *type*=*name* or *type*=*cnt* sets its *select* flag; all other cells clear their *select* flag.

2. **select-first**()
   The leftmost cell with *select* set remains unchanged; all other cells clear their *select* flag.

3. **select-successor**()
   Each cell whose predecessor has *select* set will set its own *select* flag; all other cells clear their *select* flag. The effect is to move the *select* marker to the successor of its current location.

4. **select-span-head**()
   This instruction sets *select* in the leftmost cell of the span that currently has *select* set in one of its words, and clears *mark* in the cells to the right of that point. The purpose is to locate a word whose predecessor has *attached*=**false**, where it is safe to insert new information without disturbing existing data structures.

5. **search**(n)
   This instruction sets *select* in each cell whose *mark* flag is **true** and whose *type* is *name* and *value* is n.

6. **mark-to-select**()
   Each cell to the left of the leftmost cell with *select* set will set *mark*; all other cells reset *mark*. This prepares the *mark* flags for an insertion or deletion.

7. **mark-spans()**
   This instruction sets *mark* in all cells that are in a span headed by a cell with *select* set. The purpose is to mark all the cells comprising the representation of an aggregate, so that the **search** instruction will not search in the wrong cells.

8. **insert-left**(t, v, a)
   This instruction inserts a word into the storage at a point determined by all the flag settings; the action of each cell depends on the values of its own *select* and *mark* flags. There are four cases, illustrated in Figure 3:

   (a) If the cell has *mark* set, but its successor does *not* have *select* set, then it stores the contents of its successor.

   (b) If the successor of a cell has *select* set, then the cell stores the values of the instruction operands: *type* := t, *value* := v, *attached* := a, and it sets its *select* flag and clears its *mark* flag.

   (c) If *select* is set, the cell clears *select* but otherwise remains unchanged.

   (d) If neither *select* nor *mark* is set, the cell keeps its old contents.

   The effect of this instruction is to destroy the contents of the leftmost cell (where the system keeps its available space pool), and to insert a new word just to the left of the previously selected word. The *select* and *mark* flags are left in a state that allows another **insert-left** instruction to insert another word just to the left of the newly-inserted word. Thus the interpreter can insert a vector of words by using the **select** instruction to set *select* at the point of insertion, executing **mark-to-select** to prepare the *mark* flags, and then iteratively executing **insert-left** to insert the elements of the vector, one by one.

9. **delete()**
   This instruction deletes a word. There are three cases:

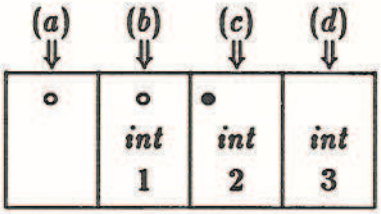
   (a) If a cell has either *select* or *mark* set, then it stores the contents of its predecessor.

   (b) If a cell has both *select* and *mark* cleared, but its predecessor has *select* set, then it sets *select* but otherwise remains unchanged.

   (c) If a cell has *select* and *mark* cleared, and its predecessor has *select* cleared, then it remains unchanged.

   The **delete** instruction deletes and reclaims a word of information by shifting a sequence of words to the right. This moves a new word from the Left Port into the available pool.

(a) Initial representation, showing the insertion category of each cell:



(b) After executing **insert-left**(*def*, 100, **true**):

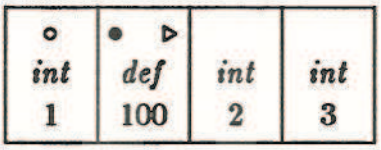


Figure 3. Example of **insert-left**(*def*, 100, **true**)

5. Implementation of lookup, update and destroy

The interpreter for an applicative language using the AAM system is a microprogram that executes in a conventional processor and issues instructions to the AAM storage unit in order to maintain its data structures. This section describes how the microprogram implements the aggregate manipulation instructions.

The AAM storage hardware returns a response to each instruction of the form $[s, t, v, a, pa]$, where each field is the logical OR of values output by each cell. Only the selected cell outputs nonzero values for these fields (except *pa*), so the response allows the processor to fetch the selected cell on each cycle. The $[s, t, v, a, pa]$ fields correspond respectively to the *select*, *type*, *value*, *attached* and *predecessor attached* values for each cell. The interpreter can read useful information by ensuring that only one cell has *select* set.

The **update** procedure (Figure 4) first allocates a new identification number for the aggregate that it is about to create. Then it selects the *def* word of the argument aggregate and prepares the *mark* flags throughout the AAM for insertion by executing **mark-to-select**. At this point there are two cases, since the representation of the

Update name n in aggregate x with new type t and new value v.

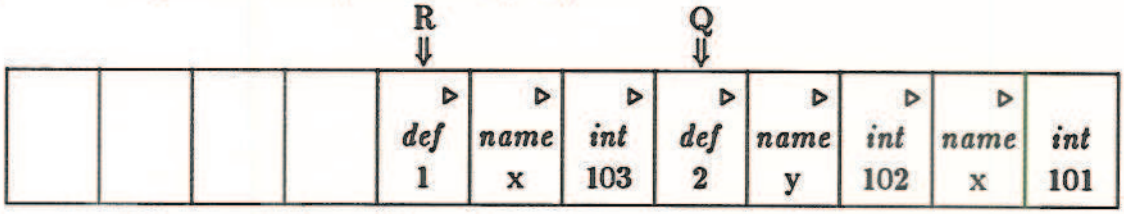
```
procedure update(x, n, t, v)
  begin
    var y : ref;
    y := allocate-id();
    select(x);
    mark-to-select();
        (sets pa to the value of attached in the cell
         whose successor has select set).
    if pa
    then
      begin
        insert-left(cnt, y, true);
        select-span-head();
        insert-left(t, v, false);
        insert-left(name, n, true);
        insert-left(def, y, true)
      end
    else
      begin
        insert-left(t, v, true);
        insert-left(name, n, true);
        insert-left(def, y, true)
      end;
    return(ref, y)
  end
```

Figure 4. update

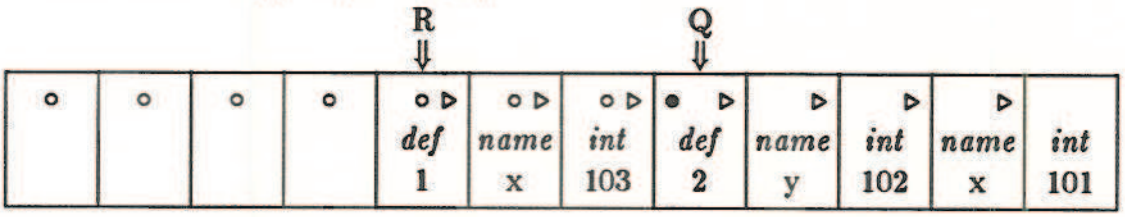
new aggregate depends on whether the word that precedes the selected word has its *attached* flag set (Section 3). The microprogram determines this by testing the *pa* field of the response to the **mark-to-select** instruction, and it then sequentially inserts the words comprising the new representation. Figure 5 shows how **update** produces the representation shown in Figure 2e.

The procedure **lookup** (x, n) (Figure 6) first selects the *def* for x and then issues **mark-spans** to set the *mark* flag in each cell that contains part of the representation of x. Each marked span must begin with either a *def* or *cnt* matching the identification

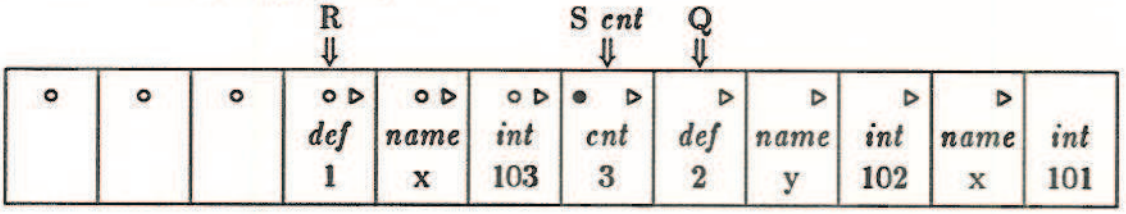
(a) Initial representation (same as Fig. 2d).



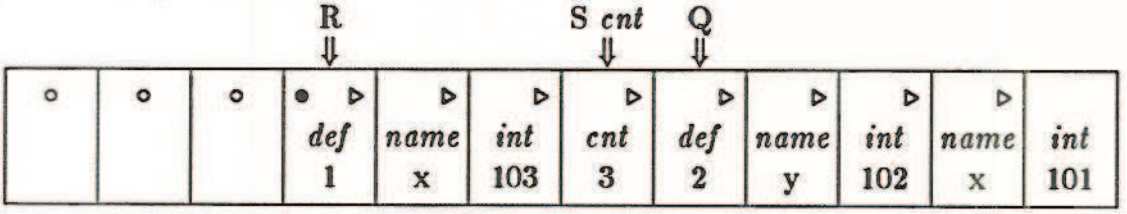
(b) $y :=$ **allocate-id()** *result:* $y = 3$
select(x) $(x = 2)$
mark-to-select() $(pa =$ **true**$)$



(c) **insert-left**$(cnt, y,$ **true**$)$



(d) **select-span-head**



(e) **insert-left**$(t, v,$ **false**$)$ $t = int, v = 104$
insert-left$(name, n,$ **true**$)$ $n = z$
insert-left$(def, y,$ **true**$)$ $y = 3$

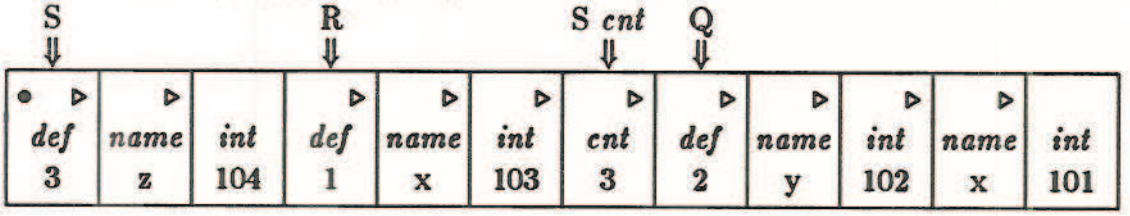


Figure 5. Example of **update**$(Q, z, int, 104)$

number of x. The microprogram next issues a **search**, which sets the *select* flag in every cell that belongs to the representation of x and that contains the name n. There may be several such cells, so the microprogram next issues **select-first** to set *select* in the *leftmost* marked cell (this cell contains the name field of the correct binding). Then the **select-successor** operation selects the value field of the binding. Finally the microprogram checks to see whether there is any cell with its *select* flag set. If not, it returns **unbound** because there is no binding for n in x; otherwise it returns the *type* and *value* of the selected cell, which constitute the value bound to n in x. Figure 7 illustrates **lookup**(S, x), using the data structures produced in Figures 2 and 5. Note that there are two extant bindings for x, but **lookup** ignores the binding belonging to R since it is not in any of the spans belonging to S.

Look up name n in aggregate x.

```
procedure lookup(x, n)
  begin
    select(x);
    mark-spans();
    search(n);
    select-first();
    select-successor();
        (sets s iff some cell has select set)
    return (if s then (t, v) else unbound)
  end
```

Figure 6. lookup

The applicative language interpreter is responsible for determining when a representation becomes garbage. It can do so using either garbage collection or reference counting – or, preferably, both. The AAM architecture presented in this paper provides no special facilities for storage management algorithms, but it can easily be augmented with powerful instructions to support both garbage collection and reference counting. Reference counting works very well with the AAM architecture, because the reference counts do not need to be stored explicitly: the tree architecture can count the number

of references to an object by associative searching in one storage cycle. This technique saves storage bits, and it also guarantees that there is no "sticky" maximum reference count value that cannot be decremented.

Once the interpreter determines that aggregate x is garbage, it must delete as much as possible of the representation of x without destroying any information that is shared with another object. A simplified implementation of **destroy**(x) appears in Figure 8. It deletes the *def* word for x, since there is no reference to it. If the word that precedes the *def* has *attached* set, **destroy** cannot delete any more words because they are shared by the aggregate preceding x. Otherwise **destroy** repeatedly deletes words until it reaches the end of the representation of x or encounters a *def* or *cnt*, whichever comes first.

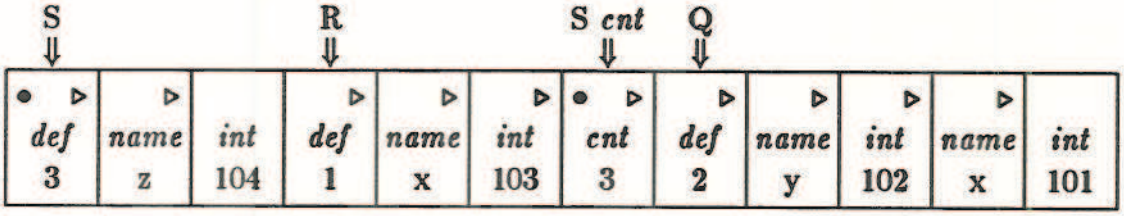
Although there is no bound on the execution time of **destroy** (unlike **lookup** and **update**, which both execute in a constant number of cycles), each iteration of the **repeat** loop in **destroy** reclaims one word of garbage. In most applications that is a good tradeoff, but [12] gives a method for incrementally reclaiming similar data structures so that all the interpreter's operations execute in bounded time.

This implementation of **destroy** does not always produce the most compact representation possible, but it is straightforward to perform several optimizations. For example, there may be redundant bindings to the same name where one of the bindings will never be used. Associative searching can detect and remove those bindings. It is also possible that an aggregate representation may contain two adjacent spans (for example, in Figure 2e, this will happen to the representation of S if R is destroyed). By setting *attached* in the first span and deleting the *cnt* that heads the second span, **lookup** could reclaim another word.

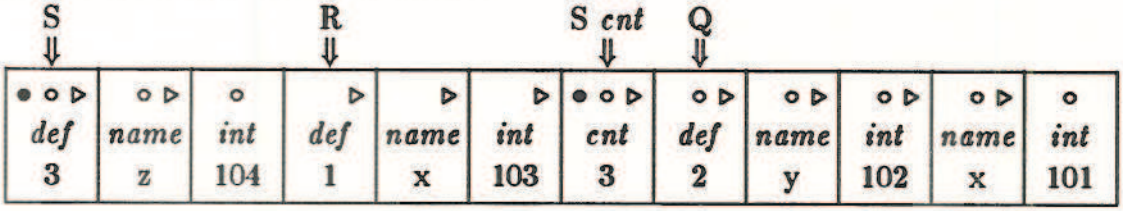
6. Hardware Implementation of AAM

Figure 9 shows the organization of the AAM storage system. The cells (square boxes) contain storage (*type*, *value*, *select*, *mark* and *attached*) and combinational logic, enabling each independently to do its part in executing the instructions. The binary tree consists of nodes of pure combinational logic, with no storage elements. Other tree architectures and algorithms are described in [18], [9] and [14]. Most of those

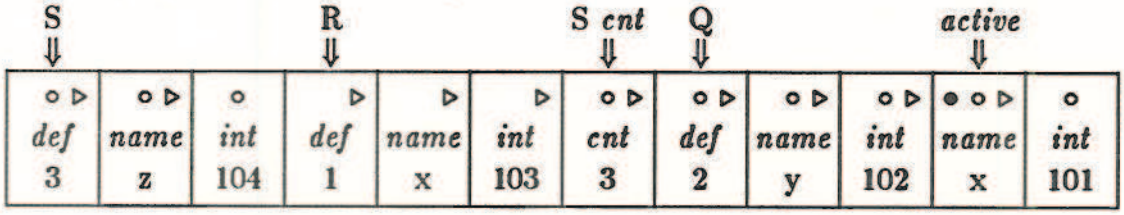
(a) **select(3)** *result:* [**true**, ...]



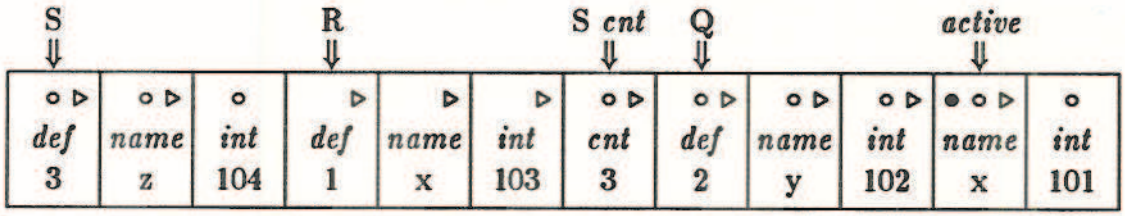
(b) **mark-spans()**; *result:* [**true**, ...]



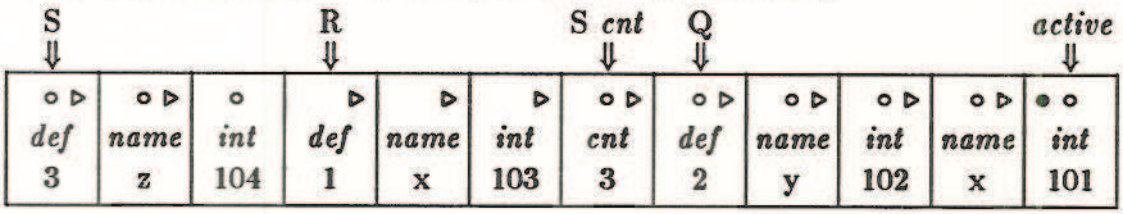
(c) **search(x)**; *result:* [**true**, ...]



(d) **select-first()**; *result:* [**true**, *name*, x, **true**, **true**]



(e) **select-successor()**; *result:* [**true**, *int*, 101, **true**, **true**]



(f) (s = **true**), *therefore* **lookup** *returns* (*int*, 101)

Figure 7. Example of **lookup**(S, x).

Destroy the portion of the representation of aggregate x that is not shared with any other aggregate representation.

```
procedure destroy(x)
  begin
    select(x);
    mark-to-select();
    repeat
      delete()
          (Sets pa to the attached value of the cell whose
            successor is selected; sets a to the attached
            value of the selected cell; sets t to the
            type field of the selected cell.)
    until
      pa  or  (not a)  or  (t = def)  or  (t = cnt);
    release-id(x)
  end
```

Figure 8. destroy

trees contain processors with clocked storage elements in every node, and they attempt to put many of the processors to work simultaneously. In a tree architecture with storage elements in the nodes, it may take as much as $(\log n)^2$ gate delay time to send information from the root to a leaf, where there are n leaves. In the AAM the gate delays take no longer than $\log n$ time.

The AAM pays a penalty, compared with RAM memory, in two important cost measures: chip area and clocking speed. Analysis of a preliminary VLSI [10] layout shows that neither penalty will be too severe. For a layout with n storage cells (we assume $n = 2^{2k}$), the chip area required is $(4n - 4\sqrt{n} + 1)$ times the area required for one cell. Thus the node combinational logic and data path connections introduce a factor of 4 penalty in chip area compared to a RAM. There is another penalty factor of about 2 to 4 because of the logic required for each cell. Although the AAM uses an order of magnitude more chip area than a RAM, it scales to large n very well. The AAM has gate delay time proportional to $\log n$ — but so does RAM memory. The chief constraint on speed is the total data path length, which is $O(\sqrt{n})$ for both AAM

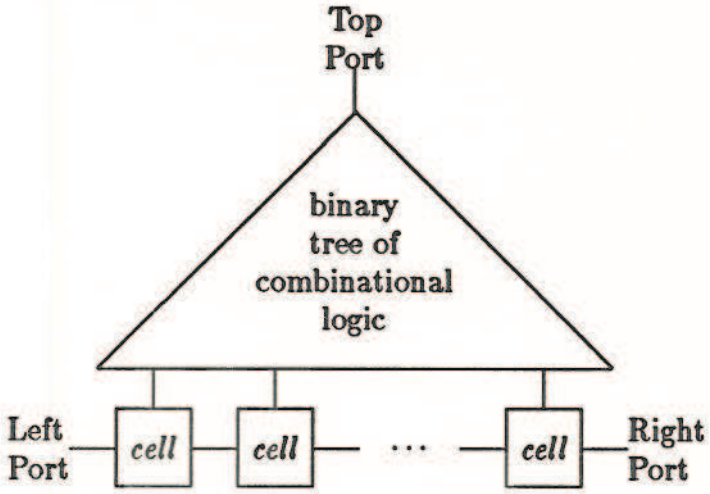


Figure 9. Organization of the AAM Storage

and RAM. In both AAM and RAM layouts the chip must broadcast information over long data paths: the AAM's nodes do this, while RAM chips have dedicated amplifiers. The AAM does computation in its broadcast and communication circuits, while a RAM does not.

The AAM organization provides several basic mechanisms that support fast execution of the instruction set:

1. Each storage cell is connected to its predecessor and successor by input and output data paths. This interconnection enables the system to shift the contents of storage in one cycle: each cell stores the contents of its predecessor or successor for a right or left shift respectively.

2. In addition to its *type* and *value* fields, each storage cell contains three control flags, *select*, *mark* and *attached*, and it also has combinational logic. These facilities allow the cells to perform independent conditional operations. For example, insertion and deletion instructions cause the cells to shift their contents conditionally, based on the values of their *select* and *mark* flags. Similarly, `search` causes each cell to compare its own contents with the instruction operands (a standard associative memory technique [3]).

3. The binary tree of combinational logic can perform global operations that depend on flag values throughout the entire set of storage cells. The **select-first**, **select-span-head**, **mark-to-select** and **mark-spans** instructions all use the tree to determine which cell flags to set.
4. The binary tree also broadcasts messages to all the storage cells. In addition to conditionally shifting words, the **insert-left** instruction must broadcast the (a, t, v) operands to all the cells, since each cell determines independently whether it will store the operands, based on the values of its own flags. This differs from conventional memories, where an address determines which storage cell must receive each message.

The hardware logic equations for the cells' combinational logic is straightforward, using multiplexors to select the values to be stored in each cell field according to the instruction operands and the cell contents. Each cell contains comparators for matching its own *type* and *value* with the t and v instruction operands. The combinational logic for the tree nodes is also straightforward for most of the instructions. For example, the **select-first** instruction requires node combinational logic similar to standard associative memory priority circuits [3]. The **mark-spans** instruction requires much more complex combinational logic, since a global analysis of all the *select* and *attached* flags is necessary to determine whether each cell is a member of a selected span.

The aggregate manipulation functions **lookup** and **update** always require a constant number of AAM storage cycles. The storage cycle gate delay time is proportional to $\log n$ for a storage containing n cells. Thus the **lookup** and **update** operations require log time with respect to *total* storage size, and they require *constant* time with respect to aggregate representation size.

7. Conclusion

This paper has described a new solution to the "aggregate update problem" for applicative languages. The solution consists of two levels: at the higher level, a microprogram maintains a data structure representation of "associative aggregates", and at the lower level a hardware storage design implements the powerful operations required by the microprogram. The associative aggregate data structures cannot be implemented efficiently on conventional architectures.

There is currently great interest in building parallel computer systems containing large numbers of processors and storage units. But there are several hard problems in designing highly parallel systems:

1. finding something useful for all the components to do,
2. programming them to do it, and
3. arranging for communications that allow the components actually to run simultaneously.

The AAM system contains a single processor and a single storage, so at the level of processor networks it does not appear to be a parallel computer. Actually the AAM is a highly parallel system, but its parallelism exists entirely within the storage. The components that execute in parallel are the storage cells and the tree nodes. Inside its storage the AAM solves the problems of parallelism outlined above:

1. The cells' parallelism (*i.e.* the shifting operations) performs insertions and deletions in linearly ordered data structures in a constant number of cycles. Conventional architectures require iteration for those operations, so conventional language implementations use linked list representations for data structures that require extensive insertions and deletions — thereby losing the advantages of linearly ordered representations. The tree nodes' parallelism (*i.e.* the searching and marking operations) performs associative searching and resolves multiple name bindings. It can also perform indexing [13].
2. The AAM storage needs to run only one program: the applicative language interpreter. Thus all users of the applicative language receive the full benefits of the AAM's parallelism without needing to program in the parallelism themselves. In addition, the user need not adhere to a restrictive style of programming in order to receive those benefits.
3. The interconnection network within the AAM fits exactly the communication requirements of the cells and nodes. All the data paths present are necessary, and additional data paths would not help.

One of the (hoped-for) advantages of applicative languages is that they will work well on highly parallel systems [1, 2, 16, 17, 19]. Hudak and Bloss [5] comment that "... 'modifying an aggregate' translates into modifying a *copy* of the aggregate, and the expense of copying large aggregates is extreme — indeed one might actually consider *limiting parallelism* in an effort to avoid copying." It is better to use parallelism to

make aggregate updating fast, but it is not clear how to do that on parallel systems containing large networks of conventional processors and storages. The AAM does just what is wanted: it uses parallelism to solve one of the implementation problems peculiar to applicative languages.

The AAM is a subset of a slightly more complex architecture that greatly improves efficiency in a number of useful algorithms [12, 13]. It implements very fast "reference counting" and garbage collection, supports interpreter data structures, and supplies a rich set of data structure operations to the user. The AAM represents lists compactly and provides efficient access (representation issues for conventional architectures are discussed in [6, 8]). A hardware implementation should support all these algorithms. It may also be possible to include many AAM's in a large scale multiprocessor.

The ultimate goal of this work is to develop computer architecture techniques to support programming languages. This is not simply a matter of finding better ways to implement existing languages, because architecture research sometimes suggests improvements in programming style or in the languages themselves. Experimentation with the AAM and its relatives has yielded several such improvements — for example, the AAM demonstrates that applicative programmers can use aggregate data structures as they wish, without worrying about inefficiencies due to recopying or shared structure representation. As we learn more about applicative programming and computer architecture, discoveries in each field will affect our ideas about the other. Current imperative languages are well-suited to the von Neumann architecture. Eventually applicative languages should be even better suited to their own architectures.

Acknowledgements

I would like to thank Cordelia Hall, Mary Kitchel, Sidney Kitchel and two anonymous reviewers for helpful comments on this paper.

8. References

1. John Backus, "Can Programming be Liberated from the von Neumann Style?", *CACM* Vol. 21, No. 8, 1978.

2. John Darlington and Mike Reeve, "Alice: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages", *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, ACM, New York, 1981.

3. Caxton C. Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Co., New York, 1976.

4. Michael J. C. Gordon, *The Denotational Description of Programming Languages; an Introduction*, Springer-Verlag, New York, 1979.

5. Paul Hudak and Adrienne Bloss, "The Aggregate Update Problem in Functional Programming Systems", *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985.

6. Robert M. Keller, "Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems", *Conference Record of the 1980 LISP Conference.*

7. David J. Kuck, *The Structure of Computers and Computations*, Vol. 1., John Wiley & Sons, New York, 1978.

8. Kai Li and Paul Hudak, "A New List Compaction Method", Research Report YALEU/DCS/RR-362, Yale University, New Haven, 1985.

9. Gyula A. Mago, "A Network of Microprocessors to Execute Reduction Languages" (Parts 1 and 2), *International Journal of Computer and Information Sciences*, Vol. 8, No. 5 (Part 1) and No. 6 (Part 2), 1979.

10. Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

11. Alan Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, CST-15-81, University of Edinburgh, 1981.

12. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, The University of Iowa, Iowa City, 1981.

13. John T. O'Donnell, "An Efficient Architecture for Implementing Sparse Array Variables", to appear.

14. Thomas A. Ottman, Arnold L. Rosenberg and Larry J. Stockmeyer, "A Dictionary Machine (for VLSI)", *IEEE Transactions on Computers*, Vol. C–31, No. 9, 1982.

15. Ravi Sethi, "Pebble Games for Studying Storage Sharing", *Theoretical Computer Science*, Vol. 19, No. 1, 1982.

16. Philip C. Treleaven, "Computer Architecture for Functional Programming", *Functional Programming and its Applications*, J. Darlington, P. Henderson and D. A. Turner (ed.), Cambridge University Press, Cambridge, 1982.

17. Philip C. Treleaven, David R. Brownbridge and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", *ACM Computing Surveys*, Vol. 14, No. 1, March, 1982.

18. Jeffrey D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

19. Steven R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.