Accumulators: New Logic Variable Abstractions for Functional Languages

Keshav Pingali*
Kattamuri Ekanadham

88-952
December 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by NSF grant CCR-8702668 and an IBM Faculty Development Award. This paper will be presented at the 8th Conference on Foundations of Software Technology & Theoretical Computer Science, December 1988, Pune, INDIA.

Accumulators: New Logic Variable Abstractions for Functional Languages

Keshav Pingali†
Computer Science Department
Cornell University
Ithaca, N.Y.

Kattamuri Ekanadham
IBM Research
T.J.Watson Research Center
Hawthorne, N.Y.

Abstract

Much attention has been focused by the declarative languages community on combining the functional and logic programming paradigms. In particular, there are many efforts to incorporate logic variables into functional languages. We propose a generalization of logic variables called *accumulators* which are eminently suited for incorporation into functional languages. We demonstrate the utility of accumulators by presenting examples which show that accumulators can be used profitably in many scientific applications to enhance storage efficiency and parallelism.

CR Classification Numbers: D.1.1, D.3.2, D.3.3, D.3.4,D.1.3

†Supported by NSF grant CCR-8702668 and an IBM Faculty Development Award.

Accumulators: New Logic Variable Abstractions
in Functional Languages

1 Introduction

Declarative languages, such as functional and logic programming languages, have received much attention lately as appropriate vehicles for programming parallel machines. Conventional imperative languages such as FORTRAN or PASCAL have sequential operational semantics based on commands that cause side-effects in a global store. These languages can be adapted for parallel machines by extending them with annotations, using which the programmer can request parallel execution in chosen parts of his program. Unfortunately, imperative languages with parallel annotations can exhibit unintended non-deterministic behavior because of races between updating and reading of storage locations. A more satisfactory alternative is to make the compiler responsible for finding parallelism. Parallelism in imperative language programs is severely limited by reuse of storage locations and parallelizing compilers enhance parallelism by eliminating such reuse of storage through transformations such as renaming, scalar expansion *etc.* [9]. Evidently, the imperative programming model encourages the programmer to reuse storage only to have the compiler eliminate the reuse! This seems rather pointless - why not begin with a programming model in which storage reuse cannot arise?

Declarative languages provide precisely such a model to the programmer. In contrast to imperative languages, declarative languages can be given an operational semantics which is naturally parallel since it is based on rewrite rules rather than on updating of a global store. The entire program is considered to be an expression that is rewritten to the answer by successively simplifying sub-expressions. Sub-expressions of the program can be rewritten in parallel and a variety of parallel rewriting strategies such as parallel innermost, parallel outermost, dataflow rule etc., have been studied extensively. Moreover, parallelism comes with a guarantee of determinacy - there are a variety of theorems such as the Church-Rosser theorem that guarantee that the final result of the rewriting is independent of the order in which sub-expressions are rewritten. Much of the current interest in declarative languages stems from this unique combination of natural parallelism and guaranteed determinacy.

One active area of research in declarative languages is combining the functional and logic paradigms[5,6]. Logic languages provide a number of computational features like logic variables and backtracking which are not present in functional languages. Of particular interest to us is the introduction of logic variables into functional languages. In a functional language, an identifier obtains its value as the result of evaluation of a single applicative expression. In contrast, a *logic variable* in logic programming languages obtains its value incrementally by the intersection of successively applied constraints. The incorporation of logic variables into an otherwise functional language provides the programmer with a powerful tool for writing elegant and efficient programs for problems such as the construction of large arrays in scientific programming [4], owner-coupled sets in database programming [11], and the coding of constraint-based algorithms such as Milner's polymorphic type deduction algorithm.

The research reported in this paper arose from our efforts to use logic variables to alleviate the so-called 'copying problem' of purely functional data structures[4]. In a pure functional language, a data structure is a value (just like an integer or floating point number) which is produced as the result of evaluating a single applicative expression. This is satisfactory when the data structure is built bottom-up (as lists are): first, the components of the data structure can be constructed, and then these components can be assembled together to produce the desired data structure. However, this does not work for 'flat' data structures, such as arrays and matrices. Constructing large arrays and matrices functionally is difficult because usually, there is no uniform rule for computing matrix elements; for example, the computation of boundary elements may be quite different from the computation of interior elements. In such situations, writing a single applicative expression for defining the entire matrix can be inefficient and the resulting program may be quite obscure[4]. An alternative is to compute the desired matrix as the limit of a sequence of matrices which differ incrementally from each other. Unfortunately, the absence of an update operation in functional languages means that each matrix in this sequence is a different value, and the construction of a matrix of size $n \times n$ may involve making n^2 copies of the matrix! Logic variables provide an elegant solution to this problem because they allow the programmer to define an array incrementally without making intermediate copies. To construct a large matrix, the programmer first allocates a matrix of the desired size, in which each element is an uninitialized logic variable. These logic variables can be bound incrementally in the program, without having to copy the entire data structure; for example, the array can be passed to two procedures, one of which instantiates variables on the boundary while the other instatiates variables in the interior. In this way, large data structures can be constructed without the copy overhead of functional data structures. This use of logic variables is similar to the use of 'difference-lists' in pure logic programming.

In Section 2 we examine the notion of logic variables and observe that the means through which they acquire values is unnecessarily limited to term unification and propose a generalization to other user defined functions. We argue that logic variables should be generalized to enable the programmer to specify any commutative and associative function (such as +, *, max, min, set insertion etc.) in place of unification for giving values to logic variables. These generalized logic variables have the flavor of objects in object-oriented languages. As with any new language construct, there are two questions that must be answered. First, is the extension useful? Second, can it be easily implemented? We believe that both questions can be answered affirmatively for generalized logic variables. They can be used to lower the storage requirements of declarative language programs without any loss of parallelism; in fact, in many problems, their use enhances parallelism. Section 3 provides many examples to illustrate that logic variables are very useful for writing standard scientific programs. Section 4 presents a formal operational semantics for a functional language with logic variables and Section 5 briefly discusses some implementation considerations. Section 6 presents our conclusions and suggestions for future work.

2 Logic Variables

In functional languages, the notion of a 'variable' is absent - one only has identifiers that are synonyms for values. An identifier is introduced through a *binding* that associates it with an expression; the identifier is a synonym for the value of that expression. The programming model does not support manipulation of identifiers. As an example, consider the following definition of a function. The curly braces denote a *let-rec* style block which is a set of local bindings followed by a return expression. When the function *pq* is invoked with an argument value for *N*, an applicative interpreter first evaluates the expression *p(N)*. The identifier *X* is then bound to the resulting value, after which the function *q* is invoked. Thus, an identifier never participates in any operation until it is replaced by a value.

$$\begin{aligned} pq(N) = \quad \{ \quad & X = p(N) \\ & Y = q(N, X) \\ & Z = Y + X \\ & \textbf{\textit{return}} \ \ Z \ \} \end{aligned}$$

This is true even under normal order evaluation. A normal order evaluator delays the evaluation of expressions until they are needed to produce the output of the program. In the program shown above, the computation of the values of X and Y would be 'delayed' until an attempt was made to evaluate $Y+X$. At that point, the expressions for Y and X would be evaluated, and the identifiers bound to their values. Under normal-order evaluation, the transformation from a name to a value involves two steps: first the name X is replaced by some sort of a descriptor that points to the computation of $p(N)$ and later the descriptor is replaced by a computed value. The intermediate form is transparent to the program, because the program can never check whether the value of X has been computed at any point - the interpreter checks this internally and prevents the intermediate form to participate in any operations, except in argument passing.

Logic languages extend this behavior even further by introducing variables as 'first-class citizens'. A variable in a logic programming language represents a place holder; therefore, it can be introduced in the program without necessarily binding it to a value. Variables are bound to values by unification performed during pattern matching of arguments in a function call. This permits textual separation of the creation of a variable from the specification of a value for it. For example, the following logic program is intended to specify the same function *pq*.

$$pq(N, Z) \quad :- \quad p(N, X), q(N, X, Y), add(Y, X, Z).$$

Invoking a goal like $pq(5,Z)$, binds N to the value 5 and instantiates two new variables X and Y, which are initially undefined. By including appropriate definitions for the functions p and q it is possible to achieve the expected sequence of events, *viz.* computing $p(5,X)$ binds X to some value and computing $q(N,X,Y)$ binds Y to some value and so on.

However, logic programs offer a lot more flexibility than is apparent in the above definition of *pq*. For instance, we can define the functions p and q very differently to create effects that are not possible to reproduce in a functional language. Consider the clauses

$$\begin{aligned} & p(N, X). \\ & q(N, 0, 0). \end{aligned}$$

Invoking $p(5,X)$ returns the unbound variable X and invoking $q(5,X,Y)$ binds both X and Y to 0. This program illustrates the fact that the behavior of a variable in logic languages is fundamentally different from anything found in functional languages: the direction of flow of information is not determined a priori and the variable can participate in unification any number of times. Note that the values bound by each unification must be consistent. For example, if a variable is bound to 5, an attempt to unify it with 6 will cause a failure of unification. This ensures that the program output is determinate, even under parallel evaluation.

Operationally, a variable is associated with some storage which undergoes state transitions as depicted in Figure 1. The initial state corresponds to the variable being unbound and the final state corresponds to the variable being *grounded* or completely defined. Actually we should imagine several final states, one for each different constant value that the variable can assume. Figure 1 shows only a prototypical part of the state diagram. A transition is caused by unification. As long as the variable is unified with other unbound variables, the state remains the same. Once it is unified with a constant value, it reaches the final state and its value cannot be changed any more. Further unifications can only reinforce that its value is what was defined. An attempt to unify it with some other value results in failure (that is, an error state) which is denoted by the symbol ⏚.

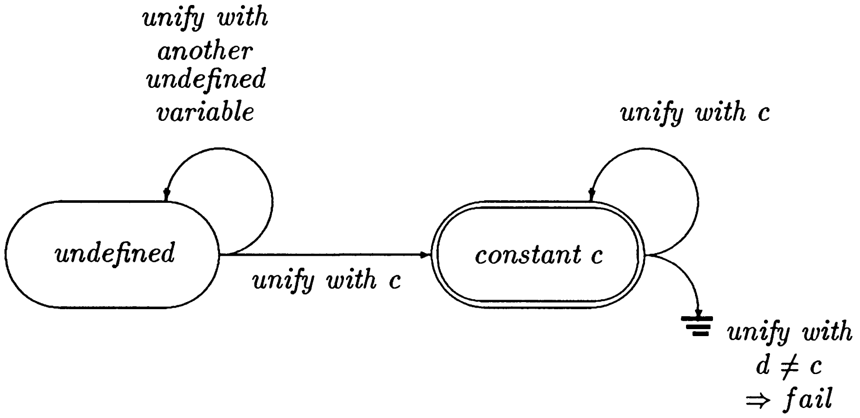

Figure 1: Prototypical state diagram of a scalar variable

How can we introduce logic variables into a functional language? In logic languages, unification is done during pattern matching of arguments in a function instantiation. Since unification may cause a 'side-effect' on the state of the variable, we prefer that the side-effect take place through an explicit command rather than implicitly during function calling. Therefore, we use the syntax

$$A = variable()$$

to introduce a logic variable and name it A, while the command

$$A \leftarrow x$$

indicates that the value x is to be unified with the variable A. These commands can occur wherever a binding can occur in the base functional language. Furthermore, this also fixes the directionality of information flow, by requiring that unification is always between a variable and a constant value. We do not consider unifying a variable with another variable.

In the next subsection, we generalize the notion of unification and its role in binding values to variables. This generalization is very different from other attempts in the literature to generalize the notion of logic variables. For instance, in constraint logic programming [10], the value of a variable is viewed as the solution to a set of constraints. Each equation is an additional constraint that potentially narrows the domain of values for the variable. This not only permits the more general notion of having constraints as values, (*e.g.*, $4 \leq X \leq 5$), but also facilitates inferencing (*e.g.*, $4 \leq X \leq 5, \;\; X \leq 4 \;\Rightarrow\; X = 4$). We do not deal with this kind of generality in this paper. Other examples of extreme generality are imperative programs, which provide explicit control over the storage of a variable, thus making the value of a variable as a function of time as well as the order of evaluation. The problems with imperative variables were discussed in the introduction.

2.1 Generalization

To generalize the notion of logic variables, we make a number of changes to the conventional logic variables shown in Figure 1. First, we permit a logic variable to undergo several state changes, rather than just 2 changes as with conventional term unification depicted in Figure 1. Second, we associate a state transition function, Φ_A, with each logic variable, A, which maps a state and an input to a new state. For conventional logic variables, this function is unification. In our generalization, the transition function can be any general function defined in the program, subject to certain constraints which we will describe shortly. A variable is created by the equation:

$$A = \boldsymbol{variable}(s, f)$$

This defines A to be a variable, whose initial state is set to s and whose state transition function Φ_A is the function *f*. Subsequently the command $A \leftarrow x$ has the effect of replacing the current state, s, of the variable A with the value of $\Phi_A(s, x)$. The state transition diagram for the generalized variable is illustrated in Figure 2.

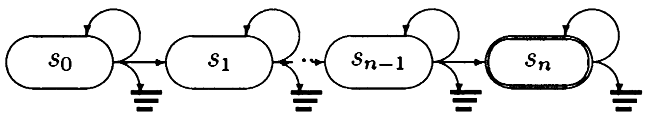

Figure 2: Generalized state diagram of a scalar variable

In order to preserve determinacy, we can use only certain functions as proper state transition functions for logic variables. As is the rule in functional languages, we will assume that the only sequencing constraints between computations are data dependencies. This means that the state transitions of a logic variable may take place in any order. To guarantee determinacy, we require that a transition function be commutative and associative as shown below.

$$\Phi(\Phi(s, x), y) \quad = \quad \Phi(\Phi(s, y), x), \quad \forall \; s, x, y \qquad (1)$$

That is, for any state s, the transitions for any two inputs x and y can be made in any order and the result will be the same. Functions like *add, multiply, min, max* are examples that have this property. For conventional logic variables, the state transition function is unification, which is also commutative and associative.

The value of the final state (if and when it is reached) replaces the variable throughout the program. Thus, we need to establish criteria to determine when a state is final. To achieve this, we keep a counter with each variable. This counter is initialized to some positive integer when the variable is created, and is decremented on each state transition. A state is *final* when the counter is 0; any attempt to make a state transition on a variable whose counter is zero is an error. In the rest of the paper, we will include the counter in the state of the variable, and write (u, v) to denote a variable whose value is v and whose counter has the integer u.

The machine implementation of a command $A \leftarrow x$ can be summarized as the following sequence of steps:

- Let (u, v) be the current state of a variable.
- Evaluate $(u', v') = \Phi_A((u, v), x)$
- If $u = 0$ and $(u, v) \neq (u', v')$ then the program fails
  Else (u', v') replaces the state of the variable.
- If $u' = 0$ then the value v' replaces the variable A throughout the program.

Note that the above steps are executed atomically, in the sense that while (u', v') is being computed, the state is not available for any other command. Later we elaborate how this can be implemented in practice.

From Figure 2, it is clear that each intermediate state is used only by the next transition that takes place. This implies that the state change can be done in the same storage without any problems. The specification of a transition function can take advantage of this and specify only the necessary changes to the state. For example if the state consists of an array and only one element of the array is changed, it should suffice to specify only that change. To facilitate this we introduce the notation of a shadow state. A shadow state, $s!$, is an uninitialized variable, identical in structure to the state s. Bindings can be made to the components of $s!$ as if it were a normal structure. When the execution of the transition function terminates, uninitialized components of the shadow are filled with copies of the state s. The following example illustrates this.

$$\begin{array}{rcl} countup((n, A), i) & = & \{\ A![i] = A[i] + 1 \\ & & \quad \boldsymbol{return} \ \ (n-1, A!)\ \} \end{array}$$

The function specifies the change of state from (n, A) to $(n-1, A!)$ where the array $A!$ is identical to A except that the i^{th} element is incremented by one.

We will call these generalized logic variables *accumulators*.

3 Examples

3.1 Example 1: Write-once Variables

Our first example is a *write-once* variable whose state transitions follow the pattern depicted in Figure 1. Its initial state is $(1, 0)$ and its final state is $(0, c)$ where c is any constant to which it can be assigned. The state transition function and the usage of the variable are illustrated by the following program segment:

$$\begin{array}{rcl} assign((u, v), x) & = & (0, x) \\ A & = & \boldsymbol{variable}((1, 0), assign) \\ A & \leftarrow & 67 \\ b & = & A + 5 \end{array}$$

The *assign* function satisfies the commutative property of equation (1), because for any $x \neq y$ the commands $A \leftarrow x$ and $A \leftarrow y$ result in error in whichever order they are executed. Notice that the evaluation of the expression $A + 5$ is automatically delayed until the command $A \leftarrow 67$ takes effect.

I-structures[4] are simply arrays whose elements are the logic variables described above. For example, we can construct an aggregate, A, of n variables, where each $A[i] = \boldsymbol{variable}((1, 0), assign)$, $\forall\ i = 1, n$. Individual array elements can be assigned values using commands of the form $A[i] \leftarrow c$.

The expressive power of these arrays can be illustrated by the inverse-permutation example: Given an array B of length n containing a permutation of integers $1..n$, compute the array A such that $A[B[i]] = i$. The following program accomplishes this. We use the informal array notation to indicate the allocation of contiguous storage, initializing each element with the specified value. For clarity, we also use the obvious iterative construct in place of tail recursion.

$$
\begin{aligned}
\textit{inverse-permute}(B, n) \quad = \quad & \{A = \boldsymbol{array}(n) \;\; \boldsymbol{of} \;\; \boldsymbol{variable}((1,0), \textit{assign}); \\
& \{\boldsymbol{for} \;\; i \;\; \boldsymbol{from} \;\; 1 \;\; \boldsymbol{to} \;\; n \;\; \boldsymbol{do} \\
& \qquad A[B[i]] \leftarrow i \;\}; \\
& \boldsymbol{return} \;\; A \;\}
\end{aligned}
$$

Building inverse permutations in a purely functional language incurs severe copying penalty, as each iteration must produce a new array by appending the new element to the old array. I-structures take advantage of the fact that each element is written only once and permit the assignments to take place anywhere in the program. Non-strictness permits A to be returned even before the elements have been computed, thus providing opportunity for some additional parallelism. One can also build open lists using these arrays. A conventional *cons* cell can be viewed as an array of 2 elements. Empty cons cells can be created and used in lists and their contents can be filled later.

3.2 Example 2: Accumulation

Consider the following definition of a tail recursive function:

$$
\textit{sum}(s, n) = \boldsymbol{if} \;\; n = 0 \;\; \boldsymbol{then} \;\; s \;\; \boldsymbol{else} \;\; \textit{sum}(s + f(n), n - 1)
$$

Assuming f is some known function, the application $\textit{sum}(0, n)$ computes the summation $\sum_{x=1}^{n} f(x)$. In a conventional evaluation scheme, the partial sums $f(n), f(n) + f(n-1), ..$ are computed and passed along the chain of recursive calls. A parallel evaluator, such as a dataflow machine, might concurrently evaluate several $f(x)$, but performs the summation sequentially. Storage for each intermediate sum is allocated dynamically and is reclaimed when its value has been consumed by the recursive call.

In contrast, we can perform the above summation using generalized logic variables as follows. The initial state of the variable is $(n, 0)$ indicating n more summations have to be done starting with zero as the initial sum. The final state is $(0, \textit{final-sum})$.

$$
\begin{aligned}
\textit{addup}((u, v), x) \quad &= \quad (u - 1, v + x); \\
\textit{sum}(n) \quad &= \quad \{A = \boldsymbol{variable}((n, 0), \textit{addup}); \\
& \qquad \{\boldsymbol{for} \;\; i \;\; \boldsymbol{from} \;\; 1 \;\; \boldsymbol{to} \;\; n \;\; \boldsymbol{do} \\
& \qquad\qquad A \leftarrow f(i) \;\}; \\
& \qquad \boldsymbol{return} \;\; A \;\}
\end{aligned}
$$

All $f(i)$ can be computed concurrently and the additions in the summation are performed in arbitrary order as and when the element values arrive. The partial sums are not circulated, but are *updated in place*, much like in imperative programs. Determinacy is guaranteed as the function *addup* satisfies the constraint of equation (1).

3.3 Example 3: Histograms

The usefulness of logic variables is greatly enhanced when the state of a variable is large, so that there is potential for concurrent operations on different parts of the state. The histogram problem illustrates this. Given a list of n numbers, each of which can be classified into one of k classes, the problem is to find the frequency of each class. An imperative program for this problem keeps the frequencies in an array, which is initialized to all zeroes. It sequentially computes the class for each number and increments the corresponding element of the frequency array. This takes advantage of the sequential nature of computation and economizes storage by performing updates in place. A functional program to solve this problem will be forced to produce a new frequency array for each number in the list, since the concept of updating is not supported. The new frequency array will be identical to the old array except for one element, which would be incremented by one. This copying overhead results in extreme inefficiency. It is possible to improve on this situation by making the compiler detect that copying is unnecessary because the updates are done in sequence[8]. However, techniques for doing this analysis are not very general, and even then, the resulting solution exhibits no parallelism. The following solution using logic variables explicitly indicates that updating can take place in the same storage.

We first observe that the frequency array cannot be modeled as an array of accumulators, where each element is like the accumulator of Section 3.2. This is because we know only the total number of accumulations for the entire frequency array and the number of accumulations for each class is not known apriori. Hence we use one accumulator whose state includes the entire frequency array. The functions *zero-array* and *classof* have meanings obvious from the context. The transition function *countup*, specifies that the i^{th} element of the frequency array A is to be incremented. The notation of $A!$ is as explained in Section 2.1.

$$
\begin{array}{rcl}
countup((n, A), i) & = & \{\ A![i] = A[i] + 1 \\
 & & \quad \boldsymbol{return}\ \ (n-1, A!)\ \}; \\
histogram(n, k, classof) & = & \{H = \boldsymbol{variable}((n, zero\text{-}array(k)), countup); \\
 & & \{\boldsymbol{for}\ \ i\ \ \boldsymbol{from}\ \ 1\ \ \boldsymbol{to}\ \ n\ \ \boldsymbol{do} \\
 & & \qquad H \leftarrow classof(i)\ \}; \\
 & & \boldsymbol{return}\ \ H\ \}
\end{array}
$$

Some functional programmers [14] have argued that the histogram problem can be solved by introducing a new array primitive *functional-accumulate* that takes a combining function and an index list (*i.e.,* a list whose elements are pairs *(i,v)* where i is an integer index) as arguments, and returns an array in which the i^{th} element is the result of applying the combining function to all values v whose index is i. In contrast to our solution, which uses a very natural generalization of logic variables, this solution seems somewhat *ad hoc.* Moreover, in many problems that we have looked at, building the index list explicitly introduces a lot of overhead in time and storage. We will demonstrate this using the particle-in-cell problem discussed next.

3.4 Example 4: Convex Hull

Jarvis' March for constructing the convex hull in a plane illustrates another situation where logic variables naturally fit in. Given a set of points in a plane, the algorithm successively finds the hull vertices as if it were gift wrapping the set of points. A basic step in this algorithm is to take a set of points and an origin and determine which of those points has minimal polar angle relative to the origin. Figure 3 illustrates this. If *H1* and *H2* are two consecutive vertices in the hull, the next vertex *H3* on the hull is obtained by scanning the set of points for minimal polar angle relative to *H2*. This is done as follows: Given the cartesian coordinates of 3 points, *p0,p1,p2*, there is a simple algorithm to determine whether *p1* or *p2* has minimal polar angle relative to *p0*. That is, there is a function *min-polar(p0,p1,p2)* that returns either *p1* or *p2* accordingly. Thus, a sequential algorithm to find *H3* in Figure 3 would be

$$a1 = \textit{min-polar}(H2, H1, p);\ a2 = \textit{min-polar}(H2, a1, q);\ a3 = \textit{min-polar}(H2, a2, r); \ldots$$

That is, starting from the extreme vertex *H1* as the current *min-polar* point, successively compare with each point *p,q,r,...* in the set, each time keeping track of the current minimum. The final point will be the hull vertex *H3*. Given two consecutive hull vertices, *H1* and *H2* and an array of points, *POINTS*, the following program uses logic variables to find the next hull vertex, *H3*:

$$\begin{aligned}
scan((n, p1), p2) &= (n-1, \textit{min-polar}(H2, p1, p2)); \\
H3 &= \{A = \boldsymbol{variable}((n, H1), scan); \\
&\quad\ \{\boldsymbol{for}\ i\ \boldsymbol{from}\ 1\ \boldsymbol{to}\ n\ \boldsymbol{do} \\
&\qquad\quad A \leftarrow POINTS[i]\ \}; \\
&\quad\ \boldsymbol{return}\ A\ \}
\end{aligned}$$

This example illustrates the flexibility of the logic variable abstraction. The state transformation function is not a simple *min* function, but computes a relation (*viz. minimal polar angle*) between the old state and the new state. Ofcourse, proving that the function *scan* satisfies the constraint of equation (1) gets more complicated. But in this case, it is clear from the application.

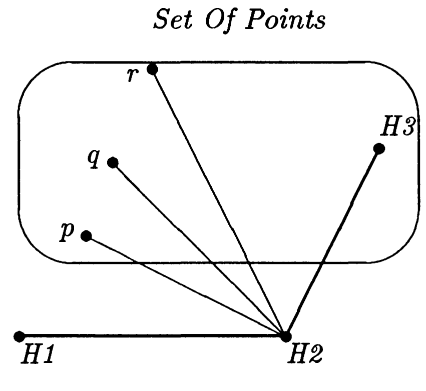

Figure 3: Computing Convex Hull in a Plane

3.5 Example 5: Particle in Cell

The particle in cell, popularly known as PIC [12], is a compute intensive problem in high energy physics that illustrates the usefulness of logic variables. The relevant parts of the problem can be abstracted as follows: A number of particles are randomly distributed over a rectangular grid of cells in a plane. Each particle is associated with a number of properties like velocity, acceleration, position within the cell *etc.* Each cell is associated with the list of particles in it, a notion of neighboring cells, charge accumulated at the cell *etc.* The presence of a particle p in cell c contributes certain amount of charge to each neighboring cell c' of c. The actual amount of charge is a function of the attributes of the particle p and the position of p relative to the target cell c'. We use the following nomenclature:

cells	= list of all cells
plist(c)	= list of particles in the cell c
neighbors(c)	= list of cells which are neighbors of cell c
contribution(p,c)	= amount of charge contributed by particle p to the cell c
CHARGE	= array, so that CHARGE[c] = the charge collected at cell c

Each cell has exactly 4 neighbors, which are adjacent to it in 4 specified directions. If p is a particle in cell c then $contribution(p, c') = 0$ if c' is not a neighbor of c. Given an initial distribution of the particles, the first step in the PIC problem is to find the charge accumulated at each cell by all the particles. An imperative program might compute the charge matrix as follows:

$$\forall \quad c \in cells \qquad CHARGE[c] = 0, \ initially,$$

$$\forall \left\{ \begin{array}{l} c \in cells), \\ p \in plist(c), \\ c' \in neighbors(c) \end{array} \right\} \quad CHARGE[c'] := CHARGE[c'] + contribution(p, c')$$

Obviously a functional program cannot perform the assignment in place. Hence it would try to specify the accumulation of charge at each cell as a single expression. For instance, one approach would be to use the inverse of the *neighbors* function. That is $c' \in neighbors^{-1}(c)$ if and only if $c \in neighbors(c')$. The idea is that all particles in cells $neighbors^{-1}(c)$ would contribute to the charge at cell c. The inverse function is simple to compute. It is the set of the 4 neighboring cells that can influence the given cell. A functional program for this problem would create a functional array, $CHARGE$, in which each element $CHARGE[c]$ is given by the function:

$$charge(c) = \forall \left\{ \begin{array}{l} c' \in neighbors^{-1}(c), \\ p \in plist(c') \end{array} \right\} \sum contribution(p, c')$$

where $\sum$ denotes the summation over the ranges specified. Although this solution performs the same number of charge computations as the imperative solution presented above, it incurs more control overhead because in this scheme each particle will be traversed 4 times, whereas the imperative solution traverses each particle only once. The solution using logic variables is presented below. Here the charge matrix is a logic variable and is updated in place using the same control structure as the imperative program, without the complication of the inverse for the *neighbors* relation. It uses the knowledge that there are altogether n particles in the system and each particle makes contributions to 4 neighboring cells. Hence the charge matrix can be bound to its value after $4n^2$ accumulations are done. The *chargeup* function performs the state change. Given the current state, a cell identifier c and a charge value q, the function increments the c^{th} element of the charge matrix by the amount q.

$$\begin{array}{ll} chargeup((u, H), (c, q)) = & \{ \ H![c] = H[c] + q \\ & \quad \textbf{\textit{return}} \ \ (u - 1, H!) \ \} \end{array}$$

$$CHARGE = \boldsymbol{variable}((4n^2, zero\text{-}matrix), chargeup)$$

$$\forall \left\{ \begin{array}{l} c \in cells), \\ p \in plist(c), \\ c' \in neighbors(c) \end{array} \right\} \quad CHARGE \leftarrow (c', contribution(p, c'))$$

Using the charge at each cell, certain equations are solved to determine the electric field and using this, the new positions of the particles are computed. Particles can move only to the neighboring cells in each step. Thus, we must construct a new *plist*. This can also be done using logic variables to update the linked list of particles in each cell. We omit the details.

3.6 Example 6: Zero of a Polynomial

Finally we illustrate another application of a logic variable in which the successive state transitions take place autonomously without any external input, until certain condition holds true. Consider the computation of the zero of a polynomial, $f(x)$, using Newton's method of approximation. Starting with an initial guess of x_0 we must compute successive refinements for the solution using the relation

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

where f' is the first derivative of f. This must be repeated until x_n is smaller than some epsilon. Hence the successive iterations need no further input. We accomplish this by the special primitive, $\boldsymbol{autovariable}(s_0, f)$, which creates a variable with the initial state s_0 and computes the sequence of states $s_1, s_2, ...$, where each $s_{i+1} = f(s_i)$. The sequence terminates when the state is final. In the following program, the variable X takes a final value when either the difference between two successive states is less than epsilon or when the number of iterations exceeds n. Notice that this effect cannot be achieved by the use of the ***variable*** abstraction, as there is no provision to execute the transition commands based on the current value of the state.

$$\begin{array}{rcl} newton(n, x) & = & \{\ y = x - f(x)/f'(x); \\ & & \quad m = \boldsymbol{if}\ \ y > epsilon\ \ \boldsymbol{then}\ \ n-1\ \ \boldsymbol{else}\ \ 0; \\ & & \quad \boldsymbol{return}\ \ (m, y)\ \} \\ X & = & \boldsymbol{autovariable}((n, x_0), newton); \end{array}$$

4 Operational Semantics

In this section, we formalize the semantics of a functional language extended with accumulators. First, we introduce a simple functional language and give a rewrite rule semantics for it. This functional language has no syntactic sugaring, so as to keep the rewrite rules simple. We then extend this language with accumulators, and extend the rewrite rules appropriately.

4.1 Syntax of a Base Functional Language

The syntax of a primitive functional language is shown in Figure 4. A program is a set of function definitions, followed by a query expression. For notational convenience, we distinguish the names of the functions from other variable names. A function expects a single argument. Multiple values can be composed into a single aggregate argument. A function body is enclosed in braces and consists of a set of equations followed by a return expression. Each equation binds a name to an expression. An expression can be a constant, a name, a conditional or the result of a binary operator applied to two expressions, in the usual manner. We omit the details of the definitions of constants, names and operators and appeal to the intuition of the reader. A function application is denoted by the function name followed by an argument enclosed in paratheses. Three other forms of expressions are given to deal with aggregate values. An aggregate is a sequence of values enclosed in angular brackets. The values in the sequence are separated by commas. A component value is selected from it using the usual subscript notation. Thus, $\langle a_1, a_2, .., a_n\rangle[i]$ gives the component a_i. The last production is a convenient abbreviation for producing aggregates. For instance, $[1\ ..\ n]\langle f, x\rangle$ gives the aggregate $\langle f(x,1),\ f(x,2), ..\ f(x,n)\rangle$. Thus, it is equivalent to an array a of n elements, which is constructed in an imperative language using a loop of the form: $for\ i := 1\ to\ n\ do\ a[i] = f(x,i)$.

$$
\begin{array}{rcl}
program & ::= & definition;\ ....\ definition;\ functionname(constant) \\
definition & ::= & functionname(name) = \{equation;\ ....\ equation \\
 & & \qquad\qquad\qquad \mathbf{return}\ expression\} \\
equation & ::= & name = expression \\
expression & ::= & constant \mid name \mid expression\ \ op\ \ expression \mid \\
 & & \mathbf{if}\ \ expression\ \ \mathbf{then}\ \ expression\ \ \mathbf{else}\ \ expression \mid \\
 & & functionname(expression) \mid \\
 & & \langle expression,\ ....\ expression\rangle \mid \\
 & & name[expression] \mid \\
 & & [expression\ ..\ expression]expression
\end{array}
$$

Figure 4: Syntax of a basic functional language

4.2 Rewrite Rule Semantics for Base Functional Language

Operational semantics can be provided for the above language using rewrite rules. Rewrite rules describe an abstract machine that maintains a state and transforms it into a final state, by repeated application of a given set of rules. Each rewrite rule specifies a set of pre-conditions and a set of actions. Whenever the pre-conditions are satisfied, the state can be transformed by executing the specified actions. Rules can be repeatedly selected and applied in arbitrary order, until no further rules can be applied. The final state is defined to be the result of the computation. Usually the rewrite rules possess certain properties which guarantee that the result is the same for all possible orders in which rules are applied. Concurrency in the application of several rules directly relates to parallel execution by a machine. Thus, rewrite rule semantics are suggestive of the potential parallelism in a program and at the same time guarantee deterministic results. Ofcourse care is needed for selecting only non-interfering rules for concurrent execution. That is, the preconditions for the rules and the effects caused by the rules must operate on disjoint parts of the state.

We now describe the state of an abstract machine. We have the usual notion of constants which are numbers, string and boolean constants, aggregates all of whose components are constants *etc. Error* is a special constant and when an expression evaluates to it, the whole computation terminates. Similarly names are identifiers defined appropriately, distinguishing variable names from function names used in the program. The state of the machine has 3 components: (1)a set of equations giving the current bindings in effect, (2)a result expression and (3)a countably infinite set of new names, that can be used during rewriting. For notational purposes, we treat a command like $A \leftarrow c$, also as an equation. The set of new names can be represented by a distinct name α and a counter. Initially, the counter is zero and each time a new name is needed, the counter is incremented and the name $\alpha_{counter}$ is used. A rewrite rule has the following form:

$$
\begin{array}{lcl}
\textit{... pattern ...} & & \textit{... new-pattern ...} \\
\textit{List of preconditions like} & & \textit{List of changes like} \\
eqn: \quad x = c & & \textit{new-eqn}: \quad \hat{x} = d \\
const: \quad c & \Longrightarrow & \textit{delete-eqn}: \quad y \leftarrow d \\
fdefn: \quad f & & \textit{mod-eqn}: \quad x = c' \\
 & & abbrev: \quad c' = c + 1
\end{array}
$$

This means that if *pattern* is found *anywhere in the result expression or on the right-hand side of any equation, but not within the arms of a conditional*, and all the stated preconditions are satisfied, then *pattern* is replaced by *new-pattern* and other stated changes are made to the state. The illustrated preconditions state that the equation $x = c$ is present in the machine state, c is a constant and f is a function name for which a definition is known. Similarly the actions state a new equation $\hat{x} = d$ must be added to the machine state, equation $y \leftarrow d$ must be deleted from the state, the equation for x in the state must be modified as $x = c'$, where c' is used as an abbreviation for $c + 1$ *etc.*

Figure 5 gives the rewrite rules for the base functional language described in Figure 4. Rule 1 simply says that if we find a pattern of two constants around a binary operator, they can be replaced by the result obtained in accordance with the rules of the operator. For example, $2 + 3$ is replaced by 5 whereas $\langle 1, 2 \rangle + 5$ might be replaced by error, which halts the rest of the execution. Similarly, Rules 2 and 3 specify how an arm of a conditional is selected. Note that the expressions in either arm of a conditional are not touched until the condition evaluates to a boolean constant. Rule 4 gives the usual substitution rule for a name. We insist that a name cannot be substituted for, until its value reduces to a constant. This avoids recomputation of the same expression in many places. Finally rule 5 says that multiple bindings to the same name result in error.

The rule for function application is somewhat complicated. We must prepare a copy of the body of the function by replacing all the names with new unique names, so that we do not get confused by naming conflicts between various instantiations of the functions. Rule 6 obtains a new name $\hat{x}$ for each name x that appears in the function definition, using the α counter described earlier. We use the notation, $|text|_{x,y,z \rightarrow \hat{x},\hat{y},\hat{z}}$ to denote the result of replacing all occurrences of the names x, y, z in *text* by $\hat{x}, \hat{y}, \hat{z}$ respectively. New equations are added to the state by renaming the equations in the function definition as shown in Rule 6 and the function application is replaced by the renamed return expression of the function.

Aggregate selection and construction are intuitive. Component selection is performed only after the index and the corresponding element reduce to constants. Similarly, an aggregate is constructed after the bounds reduce to constants.

Rules for Substitution

$\ldots\ m\ op\ n\ \ldots$	$\Longrightarrow$	$\ldots\ r\ \ldots$	(1)
$consts:\quad m, n$		$abbrev:\qquad r = m\ op\ n$	
$\ldots$ ***if*** true ***then*** e1 ***else*** e2 $\ldots$	$\Longrightarrow$	$\ldots$ e1 $\ldots$	(2)
$\ldots$ ***if*** false ***then*** e1 ***else*** e2 $\ldots$	$\Longrightarrow$	$\ldots$ e2 $\ldots$	(3)
$\ldots\ x\ \ldots$	$\Longrightarrow$	$\ldots\ c\ \ldots$	(4)
$eqn:\quad x = c$			
$const:\quad c$			
$eqn:\quad x = e1$	$\Longrightarrow$	$error$	(5)
$eqn:\quad x = e2$			

Rule for Function Application

$\ldots\ f(e_0)\ \ldots$	$\Longrightarrow$	$\ldots\ \hat{e}_{n+1}\ \ldots$	(6)
$fdefn:\ f(x_0) =$		$new\text{-}eqn:\quad \hat{x}_0 = e_0,$	
$\{x_1 = e_1; \ldots x_n = e_n;$		$new\text{-}eqns:\quad \hat{x}_1 = \hat{e}_1,\ \ldots\ \hat{x}_n = \hat{e}_n$	
$\textbf{\textit{return}}\ e_{n+1}\}$		$new\text{-}names:\quad \hat{x}_i$	
		$abbrev:\quad \hat{e}_i = \lvert e_i \rvert_{x_0, x_1, .. x_n \rightarrow \hat{x}_0, \hat{x}_1, .. \hat{x}_n}$	

Rules for Aggregate construction and selection

$\ldots\ \langle c_1, c_2, .. c_n \rangle [i]\ \ldots$	$\Longrightarrow$	$\ldots\ c_i\ \ldots$	(7)
$int:\quad 1 \le i \le n$			
$const:\quad c_i$			
$\ldots\ [i\ ..\ j] \langle f, x \rangle\ \ldots$	$\Longrightarrow$	$\ldots\ \langle \hat{x}_i, \hat{x}_{i+1}, .., \hat{x}_j \rangle\ \ldots$	(8)
$int:\quad i \le j$		$new\text{-}eqn:\quad \hat{x} = x$	
$fdefn:\quad f$		$new\text{-}eqns:\quad \hat{x}_k = f(\langle \hat{x}, k \rangle)_{, i \le k \le j}$	
		$new\text{-}names:\quad all\ \hat{x}$	

Figure 5: Rewrite Rules giving the Operational Semantics for a Base Functional Language

4.3 Syntax and Semantics for Logic Variables

Figure 6 shows additional productions used to introduce logic variables into the base functional language of Figure 4. A variable expression creates a logic variable with the specified initial state and transition function. The autovariable expression does the same, except that the state transitions take place autonomously. The equation using the left arrow symbol effects a state transition on the logic variable given on the left, supplying the expression as an argument. We permit logic variables to be components of aggregates and a selector expression could be used on the left-hand side of the arrow, to specify the transitions for the selected variable. Finally, binding to a name appended by the exclamation mark indicates delayed binding within a state transition function. The rules we give later show how this is used to achieve safe updates in place.

$$
\begin{array}{rcl}
\textit{expression} & ::= & \textbf{\textit{variable}}(\textit{expression},\ \textit{functionname}) \mid \\
 & & \textbf{\textit{autovariable}}(\textit{expression},\ \textit{functionname}) \\
\textit{equation} & ::= & \textit{name} \leftarrow \textit{expression} \mid \\
 & & \textit{name}[\textit{expression}] \leftarrow \textit{expression} \\
\textit{name} & ::= & \textit{identifier} \mid \textit{identifier}!
\end{array}
$$

Figure 6: Additional Syntax to add Logic Variables

We now specify the semantics for these constructs. The essential difference between ordinary variables and logic variables is that a logic variable cannot be substituted by its value until it reaches a final state. But at the same time, its name must be available for other operations such as parameter passing, aggregate construction, selection from aggregates *etc.* We accomplish this by introducing a new class of names called β-names which are used only by the machine internally. Intuitively, a β-name corresponds to the address of the storage in which the state of a logic variable is maintained. The state of the machine is augmented with another counter and the β-names are generated from it in the same manner as the α-names we described earlier. We assume the β-names are distinguishable from other names.

The rules concerning β-names are described in Figure 7. Rule 9 specifies how a variable is created. New storage is allocated and it is referred by a new β-name β. The storage is initialized with the state value s_0 and the transition function *f*. β-names play a dual role. They are like address constants and can be substituted just like any other constants as specified by rule 10. Names occurring on the left-hand side of a transition command can also be replaced by a β-name as shown by rule 11. Intuitively this means that the address is passed for performing transitions on the state of a variable. The second role played by a β-name is that of a regular name, except that its value cannot be substituted until the variable reaches a final state. Rule 12 specifies that a β-name can be replaced when the variable reaches a final state.

State transitions can be performed one at a time using the address β, until a final state is reached. Rule 13 prohibits further transitions from a final state. Rule 14 gives the effect of a simple transition. The transition command $\beta \leftarrow c$ causes the state $\langle n, s\rangle$ of the variable to be replaced by the value of $f(\langle n, s\rangle, c)$ in an atomic manner. The command is deleted (from the machine state),

Logic Variable Creation

$$\ldots\ \boldsymbol{variable}(s_0, f)\ \ldots \quad \Longrightarrow \quad \ldots\ \beta\ \ldots \tag{9}$$

$$\begin{array}{llcll} const: & s_0 & & new\text{-}eqn: & \beta = \langle s_0, f \rangle \\ fdefn: & f & & new\text{-}\beta\text{-}name: & \beta \end{array}$$

Address Substitution for a Logic Variable

$$\ldots\ x\ \ldots \quad \Longrightarrow \quad \ldots\ \beta\ \ldots \tag{10}$$

$$eqn: \quad x = \beta$$

Address Substitution on the left-hand side of a transition command

$$eqn: \quad x \leftarrow e \quad \Longrightarrow \quad mod\text{-}eqn: \quad \beta \leftarrow e \tag{11}$$

$$eqn: \quad x = \beta$$

Value Substitution for a Logic Variable

$$\ldots\ \beta\ \ldots \quad \Longrightarrow \quad \ldots\ c\ \ldots \tag{12}$$

$$\begin{array}{ll} eqn: & \beta = \langle\langle 0, c \rangle, f \rangle \\ const: & c \end{array}$$

No Transitions from Final State

$$eqn: \quad \beta \leftarrow c \quad \Longrightarrow \quad error \tag{13}$$

$$eqn: \quad \beta = \langle\langle 0, e \rangle, f \rangle$$

Simple State Transition for a Logic Variable

$$eqn: \quad \beta \leftarrow c \quad \Longrightarrow \quad delete\text{-}eqn: \quad \beta \leftarrow c \tag{14}$$

$$\begin{array}{llcll} eqn: & \beta = \langle\langle n, s \rangle, f \rangle & & mod\text{-}eqn: & \beta = \langle f(\langle\langle n, s \rangle, c \rangle), f \rangle \\ consts: & c, s, n \neq 0 & & & \end{array}$$

Figure 7: Rewrite Rules giving the Operational Semantics for Logic Variables

to avoid repeated application of the same command. If the transition function f is simple (like add, multiply *etc.*), then this replacement can be done atomically and we do not need any further rules. More general functions are dealt with in the next section.

4.4 Semantics for Complex State Transition Functions

Conceptually, rule 14 of Figure 7 works for any function f, as long as the computation of $f(\langle\langle n,s\rangle,c\rangle)$ and the replacement take place atomically. But if the computation of f takes a long time, then we must not tie up other computations. While the computation of f proceeds, care must be taken (1)not to initiate another state transition for this variable and (2)to replace the state by the result of f, only after all concurrent computations within f terminate. We can express these constraints through rewrite rules given in Figure 9, which use a new class of names called γ-names . Intuitively, a γ-name is just like a β-name and both stand for the address of the storage where the state of a variable is stored. However, the type of access to the storage is different for β-names and γ-names . A β-name provides access to the variable only to specify transitions and to obtain its final value. A γ-name provides exclusive access to the state of a variable for changing its value in place. Using γ-names , the above two requirements are satisfied as illustrated below.

Figure 8 shows how a state transition takes place. Let β be the address of a variable, which is currently in state s. A transition command $\beta \leftarrow x$ obtains a new name γ to refer to the current state and initiates the function f passing γ as an argument. The transition function expands into a bunch of equations, replacing the parameter name by γ. The result expression of the transition function is the shadow name $\gamma!$, which represents the new state obtained by performing the specified changes. In Figure 8, we illustrate a few changes like the first component of the state is incremented, the second component is multiplied by some thing *etc.* The shadow name $\gamma!$ replaces the function application in the equation for β. The bindings for the shadow variables are rewritten as bindings to the corresponding $\gamma!$-selectors. The value s of γ is substituted for evaluating any expressions on the right-hand sides of the equations. Finally when no more substitution for γ is possible, the updates specified by the shadow bindings can take place in the same storage, yielding a new state value s'. Until this is done, no other transition can take place for β, as the value $\gamma!$ is not considered as a constant.

$$\begin{pmatrix} \beta = \langle s, f\rangle \\ \beta \leftarrow x \end{pmatrix} \Rightarrow \begin{pmatrix} \beta = \langle f(\langle\gamma, x\rangle), f\rangle \\ \gamma = s \end{pmatrix} \Rightarrow \begin{pmatrix} \beta = \langle \gamma!, f\rangle \\ \gamma = s \\ n = \gamma[1] \\ n! = n + 1 \\ w = \gamma[2] \\ w! = w * .... \end{pmatrix}$$

$$\begin{pmatrix} \beta = \langle \gamma!, f\rangle \\ \gamma = s \\ \gamma![1] = \gamma[1] + 1 \\ \gamma![2] = \gamma[2] * .... \end{pmatrix} \Rightarrow \begin{pmatrix} \beta = \langle \gamma!, f\rangle \\ \gamma = s \\ \gamma![1] = s[1] + 1 \\ \gamma![2] = s[2] * .... \end{pmatrix} \Rightarrow \quad \beta = \langle s', f\rangle$$

Figure 8: Execution of a Complex State Transition Function

Rule 15 specifies that the storage must be replaced by the instantiation of the transition function,

General State Transition for a Logic Variable

$$\begin{array}{llcll}
eqn: & \beta \leftarrow c & \Longrightarrow & delete\text{-}eqn: & \beta \leftarrow c \\
eqn: & \beta = \langle\langle n,s\rangle, f\rangle & & mod\text{-}eqn: & \beta = \langle f(\langle\gamma, c\rangle), f\rangle \\
consts: & c, s, n \neq 0 & & new\text{-}eqn: & \gamma = \langle n, s\rangle \\
fdefn: & f & & new\text{-}\gamma\text{-}name: & \gamma
\end{array} \tag{15}$$

Abbreviation: $x[..c..] \equiv x[c1][c2]..[cn]$, where c is a list of constants $c1, c2, .., cn$.
Substitution of unevaluated γ-selector expressions

$$\begin{array}{llcl}
\multicolumn{2}{l}{...\ x\ ...} & \Longrightarrow & ...\ \gamma[..c..]\ ... \\
eqn: & x = \gamma[..c..] & &
\end{array} \tag{16}$$

γ-selector substitution on left-hand side of an equation

$$\begin{array}{llcll}
eqn: & x![..c1..] = e & \Longrightarrow & mod\text{-}eqn: & \gamma![..c2, c1..] = e \\
eqn: & x = \gamma[..c2..] & & &
\end{array} \tag{17}$$

Value Substitution only for arithmetic and logical operations

$$\begin{array}{llcl}
\multicolumn{2}{l}{...\ \gamma[..c..]\ op\ e\ ...} & \Longrightarrow & ...\ a[..c..]\ op\ e\ ... \\
eqn: & \gamma = a & & \\
const: & a & &
\end{array} \tag{18}$$

$$\begin{array}{llcl}
\multicolumn{2}{l}{...\ \boldsymbol{if}\ \gamma[..c..]\ ...} & \Longrightarrow & ...\ \boldsymbol{if}\ a[..c..]\ ... \\
eqn: & \gamma = a & & \\
const: & a & &
\end{array} \tag{19}$$

Rules to Perform Update in place

$$\begin{array}{llcll}
eqn: & \gamma![..c..] = d & \Longrightarrow & delete\text{-}eqn: & \gamma![..c..] = d \\
eqn: & \gamma = a & & mod\text{-}eqn: & \gamma = a' \\
consts: & a, d & & abbrev: & a' \equiv a,\ except\ the \\
check: & no\ other\ occurrence\ of\ \gamma & & & component\ a[..c..] \\
 & & & & is\ replaced\ by\ d
\end{array} \tag{20}$$

Release New state for next Transition

$$\begin{array}{llcll}
eqn: & \beta = \langle\gamma!, f\rangle & \Longrightarrow & mod\text{-}eqn: & \beta = \langle a, f\rangle \\
eqn: & \gamma = a & & & \\
const: & a & & & \\
check: & no\ other\ occurrence\ of\ \gamma! & & &
\end{array} \tag{21}$$

Figure 9: Rewrite Rules for updating in place

while the old state is made available through the new γ-name γ. Rules 16 and 17 specify that the γ-selectors are treated as constants and can be substituted in place of names to which they are bound. Even $\gamma!$-selector names on the left-hand side of equations can be replaced. Rules 18 and 19 specify that the values for γ-selectors (*i.e.,* old state values) are substituted only when an arithmetic or logical operation must be performed using them. Indiscriminate substitution of values for γ can result in non-determinacy. For instance, suppose we have the two equations: $x = \gamma[2]$ and $x! = 5$ and we permitted the substitution of γ-selectors any where. Applying the rules in one order will give the update $\gamma![2] = 5$ and another order will leave the second equation irreducible.

Finally, rules 20 and 21 specify the updating of the components of the state and making the state available for the next transition. These two rules have the peculiar condition to check that the γ-names are not present anywhere else in the machine state (other than their bindings). We comment on this later.

5 Implementation Considerations

The rewrite rules presented in the preceding section provide a simple graph reduction implementation. However, literal graph reduction is not particularly efficient and alternative implementations can be more efficient if some architectural aids are available. We outline some of them here. We assume that data structures are implemented in a separate memory and the structure accesses are asynchronous as is the case with a dataflow machine [3]. That is, memory and processor operate asynchronously. Read and store requests are sent as messages to the memory and the processor continues with further processing. Later, when the requests are satisfied, the memory unit sends responses as messages to the processor. There must be appropriate support to tag the messages so that a processor can relate the messages to the state of its computation. Such features – *viz.* some form of asynchronous interfaces with memory and limited arithmetic/logical operations at the memory – are becoming common with many modern multi-processors, such as CEDAR, RP3 *etc.*

5.1 Simple Transition Functions

Logic variables with simple transition functions such as *add, multiply etc.* can be implemented using a tagged memory that can perform these simple functions as part of the memory controller. For example, consider the accumulation function of Section 3.2. The storage for this variable stores the count and the partial sum. A tag bit associated with this storage indicates whether the accumulator is closed or not. The execution of a transition command results in sending a request to the memory, which examines the contents of this storage. The memory controller must increment the count, update the sum and set the tag if the count becomes zero. Use of the variable in an expression results in sending a read request to the memory. The request is satisfied if the tag indicates that the accumulator is closed. Otherwise the request is delayed by entering it in a queue associated with this storage. When an accumulator is closed, the memory controller must check to see if there are pending read requests for the accumulator and release all of them. Thus, use of a memory controller guarantees atomicity for the update operation. Depending on the transition functions, the memory controllers can get significantly complex. Tagged memory is a principal component of the MIT Tagged-Token dataflow machine [3] for implementing the I-structures. Accumulators for simple functions have been implemented in a simulator for the tagged-token dataflow machine [2].

Serialization of accumulation at the memory cells may result in some loss of parallelism. If this is a serious concern, a combining network of the kind proposed for the NYU ultracomputer [7] or RP3 can be used to perform some combing of the operations in the network itself. In fact, the accumulator construct can be seen as a way of exposing the capabilities of such a combining network to the programmer.

5.2 Complex Transition Functions

Implementing arbitrary transition functions require different support. First we observe that the rewrite rules do not render straightforward implementations. For example, rules 20 and 21 involve a global check to see that all occurrences of a given name have been replaced by its value. Checking this at run time is equivalent to the garbage collection problem and can get very hairy. Compiler analysis can help this situation greatly. Transition functions can be regulated using special syntax and the use of the old state can be tracked by the compiler. A number of schemes can be adopted to determine when it is no longer needed by the function. One can use reference counts or introduce artificial dependences to trigger certain operations when their pre-conditions are met. Or one may translate these functions into sequential threads. Alternatively, architectural support can be provided by associating a key with each memory element. Memory accesses must be accompanied by appropriate keys such as, read-final, read-intermediate, update-intermediate, update-final *etc.*

Given these complications, we are skeptical of the utility of using complex transition functions. We have included them in our discussion to demonstrate the scope of accumulators.

6 Conclusion

We have described a complete (operational) semantic framework to encapsulate the notion of incremental definition of values. Conventional functional languages insist that a value be completely specified as the result of a single expression. Logic languages provide the notion of an undefined variable and permit repeated binding of values to it, as long as they are consistent. The consistency is usually restricted to successful term unification. Consequently, once a value is defined, it remains a constant. Computing the n^{th} value in a recurrence forces us to produce all the intermediate values, as if they were needed by other computations as well. They consume memory resources and are reclaimed only as part of a general garbage collection mechanism. Imperative languages, on the other hand, perform extremely well in these situations taking advantage of the fact that the order of execution is predetermined and that a compiler can determine that the intermediate values are unnecessary and hence can be replaced by the new values of the recurrence. The logic variable notion introduced in this paper captures precisely this notion and presents a model to achieve similar implementation efficiencies when the operations involved are commutative and associative. One can look at this new notion of accumulators from three aspects: expressiveness, parallelism and efficiency.

As illustrated by the examples in the paper, accumulators arise very naturally in a variety of problems. In many problems, the use of accumulators is more natural than the use of functional primitives like *functional-accumulate* discussed in Section 3.3. However, it does open up the question of determinacy for arbitrary transition functions. A reasonable compromise may be to restrict the transition function to primitives like $+, *$, *unify* etc. which are known to the compiler as being commutative and associative.

On the parallelism aspect, accumulators improve the opportunities for parallelism. The new construct provides for combining the values in arbitrary order, if they can be generated independently. Conventional and dataflow models for parallel execution must sequentialize the combining operations. Often this sequencing requires passing feedback values from one iteration to the other in a loop and the overhead for this can be substantial [1]. Accumulators eliminate the need to circulate the values around a loop in this situation and thereby save some overhead. By delegating the arithmetic/logical operations involved in a transition function to the memory unit, the processor is freed up to do other computation to that extent.

The final aspect of efficiency is more important. The major advantage of this approach that we see is that when the accumulation is to be performed upon the elements of a structure, accumulators provide an update mechanism, which is as efficient as in an imperative program, and yet preserves the determinacy of the computation. Functional programs to solve the same problem incur either substantial copying overheads or substantial restructuring of a program to get around the problem. This is illustrated by the examples of histograms and particle in cell.

Above all, we feel that the concept of logic variables as presented in this paper bears strong resemblance to object oriented programming. The logic variables are objects, and state transitions are the methods that operate on them. The abstraction hides the representational details and provides a higher level view of the object to the rest of the program. One can confine to the analysis of the transition functions, in order to establish the validity of the operations on the object. The implementation of an accumulator naturally embodies the notions of mutual exclusion and sequentialization of certain portions of code. One can use accumulators to express these constraints at a programming level. Commutative and associative transition functions are very useful as they guarantee deterministic results. Even otherwise, these abstractions are useful for modeling non-deterministic portions of computation, such as a resource scheduler that receives requests in arbitrary order and performs state changes. We have not examined this aspect and whether logic variables provide a natural framework for solving these problems or not, is a subject for future study.

Acknowledgements: We have received useful feedback about this paper from Arvind and Nikhil at M.I.T. Accumulators were implemented in Id Nouveau by Bhaskar Guharoy. We have benefited from discussions with Alex Nicolau and John Solworth at Cornell University.

References

[1] Arvind, D.E.Culler and K.Ekanadham, The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures, To appear in: *Proceedings of CONPAR 88, at University of Manchester, UK. (September 1988).*

[2] Arvind, B.Guharoy and R.S.Nikhil, MIT Laboratory for Computer Science, Cambridge, MA (August 1987), *private communication*

[3] Arvind and R.S.Nikhil, Executing a Program On the MIT Tagged-Token Dataflow Architecture, in: *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. Springer-verlag LNCS 259 (June 1987).*

[4] Arvind, R.S.Nikhil and K.K.Pingali, I-structures: Data Structures for Parallel Computing, in: *Proceedings of Workshop on Graph Reduction, Santa Fe, NM. Springer-verlag LNCS 279 (September 1986).*

[5] Bellia,M. and G.Levi, The Relation Between Logic and Functional Languages: A Survey, in: *The Journal of Logic Programming, 3:217-236 (1986).*

[6] DeGroot,D. and G.Lindstrom, Logic Programming: Functions, Relations and Equations, *Prentice-Hall, Englewood Cliffs, NJ (1986).*

[7] Gottlieb, A. *et al,* The NYU Ultracomputer - designing an MIMD shared memory parallel computer, in: *IEEE Transactions on Computers, C-32, 175-189, February 1983.*

[8] Hudak, P., A Semantic Model of Reference Counting and its Abstraction, in: *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.*

[9] Kuck,D.J. *et al,* Dependence Graphs and Compiler Optimizations, in: *Proceedings of the 8th ACM Symposium on Principles of Programming Languages, 1981.*

[10] Lassez, J-L and J.Jaffar, Constraint Logic Programming, in: *Proceedings of* 14^th *ACM POPL conference, Munich, W.Germany. (January 1987).*

[11] Lindstrom,G., Functional Programming and the logical Variable, in: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages, 1985.*

[12] Lubeck,O.M., Los Alamos National Laboratory, Los Alamos, CA (August 1987), *private communication*

[13] Martelli,A. and U.Montanari, An efficient unification algorithm, in: *ACM Transaction on Programming Languages and Systems, vol 4, No 2:258-282 (1982).*

[14] Turner, D., *private communication*