MIT/LCS/TR-356

LOGICAL STRUCTURES FOR FUNCTIONAL LANGUAGES

Michael J. Beckerle

February 1986

Logical Structures for Functional Languages

by

Michael J. Beckerle

February, 1986

© Massachusetts Institute of Technology 1986

This research was supported in part by the Advanced Research Projects Agency under contract N000-14-83-K-0125 and in part by various grants from the International Business Machines Corporation.

MASSACUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Logical Structures for Functional Languages

by

Michael J. Beckerle

Submitted in Partial Fulfillment
of the Requirements of the
Degree of

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 1986

© Massachusetts Institute of Technology 1986

The author hereby grants to M.I.T. permision to reproduce and distribute copies of this thesis document in whole or in part.

Signature of Author Michael J Beckerle
Department of Electrical Engineering and Computer Science

Certified by Arvind
Prof. Arvind
Thesis Supervisor

Accepted by _____
Prof. Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Logical Structures for Functional Languages

by

Michael J. Beckerle

Submitted to the Department of Electrical Engineering and Computer Science
of the Massachusetts Institute of Technology on January 17, 1986
in partial fulfillment of the requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Functional Programming is frequently advocated as an appropriate programming discipline for parallel processing because of the difficulty of extracting parallelism from programs written in conventional sequential programming languages. Unfortunately, the use of Functional operations often implies excessive copying or unnecessary sequentiality in the access and construction of data structures. Logic Programming languages can use logical variables to manipulate data structures more easily; however, parallel implementations of them are not well understood.

Two new programming languages which extend Functional languages with some of the additional expressive power of logical variables for manipulation of data structures are introduced. These new languages are studied in the context of two programs which cannot be expressed efficiently in a Functional language: the flat-structure problem, and the deep-append problem. The first new language allows the flat-structure problem to be solved efficiently, but loses the referential transparency of Functional languages. The second allows the deep-append problem to be solved also, but loses the property of determinacy.

keywords: Functional Programming, Logic Programming, Interpreters, Parallelism

Acknowledgements

I would like to acknowledge the advice and contributions of my thesis supervisor, Arvind. In addition I would like to thank Ken Traub for suggesting several important organizational changes, and Suresh Jagannathan for helping with proofreading.

Many others in my research group contributed indirectly to this thesis through their thought-provoking discussions.

To Joanne

Table of Contents

Chapter One: Data Structure Manipulation in Functional Languages 5

1.1 Introduction 5
1.2 A Functional Language: Lambda 7
1.3 Programming in Lambda 8
1.4 The Flat Structure Problem 9
1.5 The Deep Append Problem 11

Chapter Two: Extending Functional Programming with Logical Variables 14

2.1 The Delta Language 14
2.2 The Need for Interpreters with Simulated Parallelism 16
2.2.1 A Quasi-Parallel Lambda Interpreter 18
2.2.2 An Example of Lambda Execution 25
2.2.3 Correctness of the Quasi-Parallel interpreter for Lambda 28
2.3 The Delta Interpreter 29
2.3.1 An Example of Delta Execution 33
2.4 Programming in Delta 40
2.5 Flat Structures in Delta 40
2.6 Deep Append in Delta 41
2.7 Input and Output from Delta Programs 46

Chapter Three: Conditional Binding of Logical Variables 49

3.1 The Eta Language 49
3.2 Operational Semantics for Eta 51
3.2.1 An Example of Eta Execution 53
3.3 Deep Append in Eta 57
3.4 Programming with Non-determinism 61

Chapter Four: Conclusions 62

4.1 Summary 62
4.1.1 Cyclic Data Structures 63
4.1.2 Run Time Errors 64
4.1.3 Demand Driven Evaluation 64
4.2 Comparison to Related Work 65
4.3 Directions for Future Research 66

References 67

Chapter One

Data Structure Manipulation in Functional Languages

1.1 Introduction

Parallel processors have great potential for increasing the speed of computation; however, the languages and techniques used to program parallel machines may be quite different from those used to program sequential processors. Common programming languages, for example, FORTRAN, C, or Pascal, have many features of sequential machine architectures visible in the language. The most troublesome feature is the notion of reusable storage locations which introduces significant synchronization overheads for parallel execution. Using a storage location more than once introduces an additional *dependency* in the program. The dependency serializes the two uses of the location to avoid unintended interference. Kuck [24] gives techniques by which some of these storage dependencies can be eliminated from Fortran programs thereby exposing parallelism in sequential programs. However, the complexity of compilation is increased dramatically and for many programs, only a fraction of the potential parallelism is exposed.

Functional programming languages have been advocated by many researchers as ideally suited for execution on parallel processors because they have no notion of a store so that unnecessary dependencies cannot be expressed. In Functional languages variables always represent values and they cannot be used to represent locations of a store in assignment statements. Some parallel architectures have been designed specifically for the execution of functional languages [14, 20, 26, 34, 15].[1] In addition, complete functional languages now exist which support desirable modern programming techniques like higher-order functions, data abstraction, and type inference [9, 36]. Unfortunately, many applications programs can only be expressed in a seemingly awkward or inefficient manner as functional programs. In particular, it is difficult to manipulate arrays and to append to data structures.

[1] Initially, *Dataflow* processors [15, 2] were intended to execute functional languages also; this work is part of an ongoing effort to extend the generality of languages executable on Dataflow processors.

Logic Programming languages [23] share some of the properties of Functional Languages in that they also have no concept of a store. Moreover, it is easier to express the manipulation of data structures in Logic languages because of the properties of *logical variables.* In a logic program, a variable need not be introduced as the value of a computation, but rather can be introduced without a value, its value to be determined through constraints placed on it by the rest of the program. Unfortunately, Logic languages seem difficult to implement and there is no wide agreement about architectures or algorithms for their parallel execution [10, 38, 37, 33, 18, 6, 13]. Also, inclusion of modern techniques like higher-order functions, or data abstraction into Logic languages is still a subject of current research [17, 42]. Logic languages are continuing to evolve and have not yet reached a mature stage of development. This makes the design of appropriate execution architectures somewhat premature.

This thesis deals with the question of whether the behavior of logical variables from logic languages can be added to the functional paradigm to yield a hybrid language having more expressive power than functional languages. Such a language would be able to manipulate arrays and append to data structures as easily as a logic language, yet would maintain most of the other features of functional languages. The answer to this question seems to be yes, depending on one's goals and expectations. Functional languages have *referential transparency* [35], a property which contributes much to the simplicity and semantic elegance of functional programs. Functional programs also have the useful property of being determinate, and correct parallel implementations of them must preserve this determinacy. This thesis will present a functional language and two extended languages each derived by adding a feature of logic programming having to do with logical variables. While these extensions add expressive power, as each feature is added, some property of the functional language is lost: first referential transparency, then determinacy. On the other hand, the extended languages will both have the original functional language as a subset; therefore, all the powerful features of functional programming —such as higher-order functions— are still available. All the languages presented are pedagogical in nature; they are for illustrating the expressive power only, and should not be misinterpreted as finished language designs.

The investigation will begin with a Lambda-calculus based functional language, which we will call *Lambda*. The extended languages will be called *Delta*, and *Eta*. All three languages will have a common Lisp-like syntax. The benefit of this is that this syntax is easily distinguished from the algorithmic-style language we will use to present the interpreters for the languages. Two programs, *inverse permutation* and *tree append*, will be written in each of the three languages; these programs are intended to exercise the data-structuring facilities of the languages, and will highlight the additional expressive power of the extended languages. In addition to these specific programs we will also look at how the extended language features help with I/O, and with programming using non-determinism.

1.2 A Functional Language: Lambda

The syntax of our functional language, Lambda, is given below.

Identifiers $= I =$ **a**, **b**, **c**, **x**, **y**, **factorial**, **apples**, etc.
Constants $= C =$ **1**, **2**, **3**, ... **+**, **-**, **=**, **>**, **nil**, **true**, **false**, ... and other constants.
Expressions $= E = C \mid I \mid S \mid \lambda I.E \mid E_1\ E_2 \mid$ **+** $E_1\ E_2 \mid$
if $E_1\ E_2\ E_3 \mid (E)$
Sugarings $= S =$ **(let** $((I_1\ E_1)\ (I_2\ E_2)\ \ldots\ (I_k\ E_k))\ E) \mid$
(letrec $((I_1\ E_1)\ (I_2\ E_2)\ \ldots\ (I_k\ E_k))\ E) \mid$
$(\lambda\ (I_1\ I_2\ \ldots\ I_k)\ E)$

Expressions of the form $(E_1\ E_2)$ are called *applications*. The first expression of the sequence, is called the *rator*. It is assumed to be a function to be applied to the second expression, called the *rand*, which is the argument. As is customary in the Lambda calculus, $E_1\ E_2\ \ldots\ E_k$ is the same as
$((\ldots((E_1\ E_2)\ E_3)\ \ldots)\ E_k)$.
i.e., application associates to the left. Expressions of the form $(\lambda \mathbf{x}.E)$ are called *abstractions*. Again, by the usual conventions of Lambda calculus, the scope of the dot extends as far to the right as possible, and parentheses are used when necessary to make the grouping of expressions unambiguous. Intuitively, abstractions are the expressions used to describe user-defined functions. The category S contains "syntactic sugarings" to provide additional Lisp-like syntax. Thus:

(λ (x y ... z) E)
is equivalent to
(λx.(λy. ... (λz.E) ...)).
`Let` is syntactic sugar for: ((λ (I_1 I_2 ... I_k) E) E_1 E_2 ... E_k), and `Letrec` is used to create recursive definitions in a manner similar to `let` [19]. Recursion can be modeled in lambda calculus by using self-application or the Y-combinator [30].

This language is effectively Lambda calculus extended with booleans, integers, primitive functions on the integers, conditional branch, and and equality predicate which determines if two integers or booleans are equal. We assume that the reader is familiar with functional programming and omit a detailed operational description of this language. In our informal discussion, we use a call-by-value execution.

1.3 Programming in Lambda

We now turn to programming in Lambda, and analyzing the expressive power of functional programming languages. There are no primitives for data-structuring in Lambda, but they can be easily modeled using the already existing features. For example, tuples of any fixed size can be implemented using higher-order functions:

```
(let ((four-tuple
       (λ (x1 x2 x3 x4)
          (λ (index)
             (if (= index 1) x1
             (if (= index 2) x2
             (if (= index 3) x3
             (if (= index 4) x4
             (if (= index 0) 4 ; returns the length
       ))))))))
  ;; now to use the four-tuple
  (let ((tup (four-tuple 2 3 5 7)))  ;; creates the tuple
    (+ (tup 2) (tup 4))))            ;; accesses the tuple
  ;; should result in an answer of 10.
```

`Four-tuple` is a higher-order function, and A call to it with four arguments returns a *function* which when applied to the integers 1, 2, 3, or 4, returns the respective original argument. Applying it to 0 yields 4, the length of the structure. It should be clear that the familiar *cons*, *car*, and *cdr* data-structure of Lisp is also easy to implement in Lambda. This technique works because function values are often represented as *lexical closures*, that is, an

ordered pair containing the function definition and an environment which contains the values of the free variables used in the function definition. Producing a function value usually implies allocation of storage to extend the environment, so it is not surprising that data structures can be modeled using higher-order functions.

An important restriction to notice about these data-structures is that all the contents of the structure must be supplied at the time of the creation of the structure.[2] It is not possible to first allocate an empty tuple, and then use indexing to *fill in* the elements as one could in an imperative programming language; nevertheless, once a structure has been created it can be indexed freely.

Since we can model tuples using closures in this manner, it is reasonable to make tuples a part of the language by providing four forms for manipulating them. (**tuple** E_1 E_2 ... E_k) will be used to create a k-tuple. (**select** E_1 E_2) will choose the element of E_2 stored at index E_1. (**replace** E_1 E_2 E_3) will produce a new tuple by copying E_2, except at index E_1, where it will store the value of E_3 instead. Finally, (**tuple-length** E_1) will return the length of a tuple as an integer. We will assume that **tuple** and **replace** operations take O(k) time and space; that is, their complexity grows with the size of the tuple being manipulated. **select** and **tuple-length** will be assumed to take constant time.

1.4 The Flat Structure Problem

To discuss the limitations of Lambda, the first program we will consider is *inverse permutation.* This program is designed to test the ability to manipulate arrays or *flat structures* in a programming language. The problem is defined as follows:

Input: An array, A, of length k of integers.
Each element, A[i], contains one of the integers 1,2,...k.
No two elements contain the same integer.
Output: An array, B of length k, where B[i] = A[A[i]] for i = 1,2,...k.

A program for performing this in Lambda is:

[2] In lazy functional languages, a program to compute each element must be supplied at the time of creation of the structure. In either case, something is associated with each element of the structure.

```
(letrec
;------------

  ((iterate-loop (λ (index A B)
                   (if (> index (tuple-length A)) B
                       (let ((nextB (replace index B (select (select index A) A))))
                         (iterate-loop (+ index 1) A nextB)))))
;------------

   (inverse-permute (λ (A) (iterate-loop 0 A A))))
;------------

  (inverse-permute (tuple 2 3 5 4 1)))  ;;; perform the algorithm.
;;; the answer is the tuple 3,5,1,4,2.
```

This program consists of two function definitions, **iterate-loop** and **inverse-permute**. The **iterate-loop** routine is written recursively since we have no iteration construct in our language; during each recursion, one element of the result tuple is determined according to the specification above. The program **inverse-permute** simply calls the function **iterate-loop** to do the work. A simple analysis of **iterate-loop** shows that the behavior of this program is quite poor. Each time the function recurses, **replace** is called once, involving O(k) work for input of size *k*. The function recurses *k* times, so $O(k^2)$ time and space are used by this program, assuming that storage is not recycled by any garbage collection mechanism.[3]. Of course most of the storage is easily reclaimed in functional languages by a simple reference count scheme; however, the inability to update an array efficiently takes this simple algorithm from O(n) time to $O(n^2)$ time. It should be clear that the common use of "flat" tuples for vector-like data structures in numerical applications, will not be efficient in pure functional languages simply because of the cost of updating these structures.[4]Some researchers advocate a tree representation even for vector-like data structures to reduce the overhead for **replace** from O(n) to O(log n) [1].

---

[3]It is reasonable to propose that a compiler perform automatic program transformations to reduce the kinds of inefficiencies shown here. The compiler would convert the functional program into an equivalent imperative program which is more efficient [39] [5]. The objection expressed here to functional programs is not that they cannot have efficient and effective compilers, but only that the language does not allow one to express programs which are as efficient as one would like.

[4]There are several techniques for improving the efficiency of **replace** in functional languages. An important one is keeping reference counts of the number of outstanding references to a structure. If the reference count of a structure is exactly 1, then it can be updated in place without copying the contents into a new structure. Unfortunately the worst case time for the algorithm is unchanged, and in the inverse permutation algorithm, such an optimization would only be possible if the execution takes place completely sequentially. Parallel access to the structures involved implies that the reference counts will generally be greater than 1.

1.5 The Deep Append Problem

The deep append problem is motivated by the suggestion that tree-like representations of data-structures be used for functional programming. The program we will use to illustrate the problem is called *tree append.* The problem is:

Input: A list of integers, each distinct.
Output: A binary search tree of these integers produced by appending the integers one at a time to the tree.

The key restriction of this definition is that this is an "on-line" problem; that is, we can think of the list of integers as being produced slowly, and the algorithm must append each integer to the tree as soon as it becomes available. In other words, the point of the program is to express appending, not to express construction of a whole from a collection of the individual elements. To write this program in the Lambda language, we will assume that we have a tuple constructor called **make-node** which makes a node of a tree containing a **left-subtree**, **right-subtree**, and **node-value**, where the corresponding field of a node is selected using a function with the same name. We will also assume there is a distinguished constant **nil** which is recognized by the predicate function **null?**. This will be used to represent the empty tree, and the empty list. The list of integers will require the familiar **cons**, **car**, **cdr**, and **list** list operations.

```
(letrec
;--------
   ((append-integer (λ (int tree)

        ;; appends an integer to an existing tree.

        (if (null? tree) (make-node nil nil int) ;; add the integer at a leaf

             (if (< int (node-value tree))      ;; else compare to current node value

                  (make-node (append-integer int (left-subtree tree)) ; append to left
                             (right-subtree tree)
                             (node-value tree)))

                  (make-node (left-subtree tree)
                             (append-integer int (right-subtree tree)) ; or to right
                             (node-value tree)))))

;--------
```

```
    (tree-append (λ (list-of-ints tree)
        ;; appends elements of list one at a time.
        (if (null? list-of-ints) tree   ; done, so return the finished tree.
            (tree-append (cdr list-of-ints)   ;; append one and recurse
                  (append-integer (car list-of-ints) tree))))))
;---------

 (tree-append (list 4 3 5 2 6) nil))  ;now try it out.
```

The program consists of two routines: **tree-append** and **append-integer**. **Tree-append** recurses once for each integer to be appended to the tree, calling **append-integer** each time. **Append-integer** just recursively descends the tree comparing the integer to be appended with each value, and descending the left or right subtree depending on the outcome of the comparison. The important property of this algorithm is that it must call **make-node** once for each node on the path from the root of the tree to the leaf where the integer is inserted, potentially copying a large number of nodes as is shown in figure 1-1. This copying seems quite expensive, and makes this algorithm require at least O(n log n) storage with a worst case of $O(n^2)$, instead of O(n). It seems that in general, functional programs will require more storage than imperative versions of the same algorithm. This inefficiency seems to be a high price to pay for a language that obeys the single-assignment rule.

We have now looked at a simple functional language, and how data-structures are modeled in it. Two programs were used to point out particularly troublesome aspects of data-structure manipulation. The next section of the paper will show an extended language, *Delta*, and compare its performance on these same two example problems.

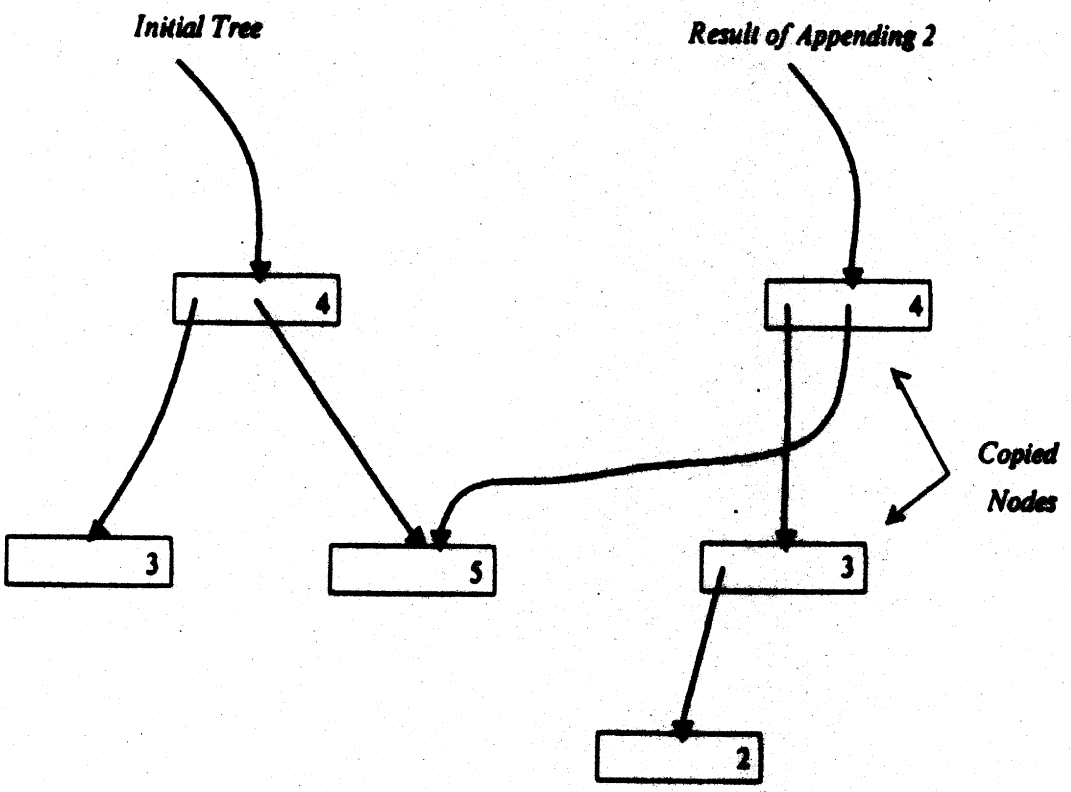

Figure 1-1: An integer being appended to a tree in the tree-append program.

Chapter Two

Extending Functional Programming with Logical Variables

2.1 The Delta Language

Functional language Lambda can be extended to include some features of Logic Programming languages. Languages such as Prolog have many attractive properties, such as pattern-driven invocation, or automatic backtracking, but for our purposes the feature of interest is variable binding by unification. This will extend the expressive power of our language in a manner still consistent with the single assignment principle.

Logic Programming languages have the ability to introduce identifiers which do not stand for values. This property is inherited from the behavior of the existential quantifier in logical formalism: $\exists x \ni P(x,y) \wedge Q(x)$ This notation introduces an identifier x and asserts predicates which must be true for some x. For example: $\exists x \ni x = 5$ is rather trivial; moreover, $\exists x \ni x = 5 \wedge x = 7$ clearly does not have any solution, and $\exists x \ni x = \langle z,w \rangle \wedge z = 5 \wedge w = f(z)$ requires x to be a pair $\langle 5,f(5) \rangle$. This property of introducing an identifier, and later constraining its value will be useful for enhancing the power of our language to deal with data structures, and it is this origin in logic that prompts the title "Logical Data Structures".

The extended language will have very similar syntax to the functional language of the previous section:

Identifiers = I = **a**, **b**, **c**, **x**, **y**, **factorial**, **apples**, etc.
Constants = C = **1**, **2**, **3**, . . . **+**, **-**, *****, **>**, **nil**, **true**, **false**, . . . and other constants.

Expressions = $E = C \mid I \mid S \mid X \mid \lambda I.E \mid E_1\ E_2 \mid +\ E_1\ E_2 \mid$
if $E_1\ E_2\ E_3 \mid (E)$

Sugarings $= S =$ (**let** $((I_1\ E_1)\ (I_2\ E_2)\ \ldots\ (I_k\ E_k))\ E$) |
(**letrec** $((I_1\ E_1)\ (I_2\ E_2)\ \ldots\ (I_k\ E_k))\ E$) |
$(\lambda(I_1\ I_2\ \ldots\ I_k)\ E)$

Extensions $= X =$ (**new**) | (**do** $E_1\ E_2\ \ldots\ E_k$) | (**==** $E_1\ E_2$)

This syntax is identical to the language Lambda except for the category X of extensions. Three constructs have been added to the language. The first, (**new**), will be used to create *unbound variables*. For example, $((\lambda x.E)$ (**new**)) introduces x as an unbound variable for the scope of the body E.

(**do** $E_1\ E_2\ \ldots\ E_k$) will be used to evaluate forms which constrain unbound variables. The expressions E_1, E_2, ... up to E_{k-1} are evaluated for their *effect* on unbound variables. Any values they return are ignored; the value returned by a **do** form will be the value of the last sub-form, E_k. **do** is intended to be used in conjunction with the == operation. (== E_1 E_2) will implement Delta's primitive subset of variable binding by unification, a kind of benign side-effect. The == operator should be read as "equate". Equate operators force the results of two computations to be equal. If one computation produces an unbound variable, via the (**new**) feature, == can be used to give it a value by introducing the constraint that this unbound variable have a value equal to that of another expression. This is different from an imperative assignment since == will succeed only on two unbound variables, one unbound variable and one value, or two equal values. No read-write race can occur in Delta because one can never use an unbound variable for any computation; it must be bound first. In addition, once a variable becomes bound to a value, that value can never change. The effect of == is simpler than unification since there is no recursive unification of data-structures or occurs check. Specifics on the interpretation of the equate operation as well as (**new**) will be deferred to the operational semantics given later.

It is important to note that the existence of a feature like (**new**) violates the property of *referential transparency* which existed in functional languages. Each appearance of (**new**) is meant to create a unique new unbound variable. and since they can occur in definitions of recursive functions, an arbitrary number of them can be created by any program.

Referential transparency is an important property of functional programs since it allows any expression to be replaced by an equivalent. For example, the following two programs in Lambda are equivalent:

```
(let ((x (+ y y)))
  (* x x y z))
(* (+ y y) (+ y y) y z)
```

The variable `x` in the first expression was replaced by its value `(+ y y)`. Referential transparency makes it possible for these two programs to be shown equivalent, and is very useful in program transformation. Delta programs are not referentially transparent, and it is easy to exhibit a program showing this:

```
(let ((x (cons (new) (new))))
  (do
    (== (car x) 6)
    (== (cdr x) 6)
    x))
```

In this code the variable `x` is introduced representing a "cons-cell" with unbound components. Then equate is used to non-locally constrain the car and the cdr of the cell, and finally the cell is returned. All the uses of the variable `x` must refer to the *same* value, i.e., an identical object. We cannot substitute the form `(cons (new) (new))` for `x`, since doing so would make the equate operators ineffective. Clearly, manipulation of Delta programs will require far more care than that of purely functional programs.

2.2 The Need for Interpreters with Simulated Parallelism

Our next goal in this thesis is to provide a concrete operational semantics for Delta so that questions of precisely how `(new)`, `==`, and `do` work can be answered. Unfortunately, this is a non-trivial task which motivates a brief digression. Consider that a *term rewriting* system can be used as an operational semantics for a language. In a term rewriting system, there is a set of rules for rewriting expressions. An expression which can be rewritten by one of the rules is called a *redex*, and expressions are rewritten using the rules until some *normal form* [21, 7] is obtained. A normal form is an expression which contains no redexes, and is what we would like to use as the "answer" to a computation. Given that an expression contains several redexes which can be rewritten, a *computation rule* determines which redexes are reduced during each step of the rewriting process. From this point of view, both

Lambda and Delta exhibit the *Church-Rosser* property [30]. This is equivalent to saying that the languages are *determinate*, a well known property of functional languages, and one which we will discuss for Delta in a later section. The Church-Rosser property says that when a normal form exists it is unique.[5] In other words, the use of different computation rules can not lead to different normal forms. We know that an expression in our source language may not have a normal form, since all our languages are capable of representing unbounded computations. Moreover, some computation rules for reducing expressions may find normal forms when other rules do not terminate. Klop [21] classifies a computation rule as *normalizing* if it is guaranteed to find a normal form when one exists. In this framework, Functional languages based on the Lambda calculus have many normalizing computation rules including *normal order reduction*, a sequential rule which always reduces the leftmost redex of an expression.[6] Luckily, the parallel reduction rule, which says to reduce all redexes simultaneously during each step, is also normalizing. The key point here is that there is at least one sequential computation rule which works for functional languages; hence, an operational semantics for a functional language can be given by a sequential term rewriting system. In other words, a simple sequential interpreter can be written for functional languages.

An operational semantics for Delta is harder to achieve. In fact, there is no sequential computation rule for Delta which is normalizing; hence, our operational semantics must be some kind of parallel reduction system. An expression in Delta which has no simple sequential interpretation is:

```
(let ((x (new))     ;introduce unbound variables x and y
      (y (new)))
  (+ (do (== y 6)  ; constrain y
         (* x x))
     (do (== x 8)  ; constrain x
         (- y y))))
   ;; the answer should be 64
```

Informally, it is easy to observe the non-sequential nature of this expression. First **x**, and **y**

---

[5] The *Eta* language, which is introduced in chapter 3, does not exhibit the Church-Rosser property.

[6] The restriction to functional languages based on lambda calculus is intended to rule out *non-sequential* functions.

are introduced as unbound variables. At this stage, we must next perform one of the equate operators in (== y 6) and (== x 8). If we chose to evaluate the first of the **do** expressions, then we will not perform the (== x 8) which is needed to give a value to the expression (* x x), so we will not be able to reduce the whole expression. By symmetry, we cannot chose to reduce the second of the **do** expressions either. To be able to reduce the expression completely we must be able to reduce parts of both **do** expressions alternately. One way to capture the notion of parallelism in execution is to note that if two reductions can occur in any order then they can occur in parallel; therefore, any reduction rule which can reduce this expression must be a parallel reduction rule.

2.2.1 A Quasi-Parallel Lambda Interpreter

To give an operational semantics for Delta we need an interpreter which simulates a parallel execution. The structure of such a *quasi-parallel* interpreter is complex. To avoid confusion, we will first describe a quasi-parallel interpreter for the Lambda language. This will illustrate how the parallelism is simulated only. Afterwards we will modify the quasi-parallel interpreter as an operational semantics for Delta.

A simple sequential interpreter for Lambda excluding syntactic sugaring is given below:

$$W(\mathbf{x}) \Rightarrow \mathbf{x}$$
$$W(\lambda \mathbf{x}.E) \Rightarrow \lambda \mathbf{x}.E$$
$$W(E_1\ E_2) \Rightarrow \textit{let}\ \alpha = W(E_1)\ \textit{if}\ \alpha = \lambda \mathbf{x}.E\ \textit{then}\ W(E[E_2/\mathbf{x}])\ \textit{else error}$$
$$W(+\ E_1\ E_2) \Rightarrow \textit{let}\ \alpha = W(E_1),\ \beta = W(E_2)\ \textit{if}\ \alpha \in N\ \textit{and}\ \beta \in N\ \textit{then}\ \alpha + \beta\ \textit{else error}$$
$$W(\mathbf{if}\ E_1\ E_2\ E_3) \Rightarrow \textit{let}\ \alpha = W(E_1)\ \textit{if}\ \alpha = \mathbf{true}\ \textit{then}\ W(E_2)\ \textit{else}\ W(E_3)$$

In this interpreter, the clause for interpreting (+ E_1 E_2) should actually be thought of as a clause schema; all binary operators in the language are implemented in an analogous fashion. The notation $E[V/\mathbf{x}]$ is used here to denote substitution of the value, V for the

symbol **x** with appropriate renaming of identifiers so that correct lexical scoping is preserved.

This interpreter reduces expressions in Lambda into weak, head-normal forms [3], but only if these forms are individual symbols or abstractions. In the Lambda calculus, a weak, head-normal form is a form where the rator of the leftmost application is not an abstraction, and is not convertible to an abstraction. For our language, Lambda, we add to this definition that the leftmost application is not (**if** E_1 E_2 E_3) or (**+** E_1 E_2) since these forms can always be reduced further.

We will now define an interpreter which produces nearly the same answers as the sequential interpreter above, but which executes with simulated parallelism. It will differ in its termination properties only. The interpreter's *state* will consist of an *activity queue*, and a *store*, and the interpreter will be described as a state-transition function, **M**. The following equations are definitions of the various objects and functions used by the interpreter, **M**:

Integers	$N = \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots.$
Values	$V = I \mid \text{closure}(E, \rho) \mid N$
Identifiers	$I = \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{x}, \mathbf{y}$, etc.
Expressions	$E = I \mid \lambda \mathbf{x}.E \mid E\ E \mid +\ E\ E \mid \text{if}\ E\ E\ E \mid (E)$
Locations	$Loc = 0, 1, 2, \ldots$
Environment	$\rho = I \rightarrow (Loc + I)$
Store	$\sigma = Loc \rightarrow (V + UNBOUND + Loc)$
Activity	$Act = \langle INTERP, E, \rho, Loc\rangle +$ $\langle APPLY, Loc, Loc, Loc\rangle +$ $\langle +, Loc, Loc, Loc\rangle$ $\langle BRANCH, Loc, E, E, Loc\rangle$
Activity-Queue	Act^*
State	$Act^* \times \sigma$
M	$State \rightarrow State$

The environment, ρ, is a mapping of identifiers to the locations they represent. As in the sequential Lambda interpreter above, unbound identifiers are considered to be constants, so

the environment will map unbound identifiers to themselves. Applying the environment to an identifier, $\rho(I)$, returns the location which that identifier represents. Substitutions are used to indicate extensions to the environment, $\rho[L/I]$. An initial empty environment will be denoted by ρ_0.

The interpreter will make use of *environments*, ρ, to represent the substitutions of values for bound variables; therefore, the set of *values* includes lexical closures of expressions and environments built by the operation **closure**(E,ρ).

The *store*, σ, behaves like a memory which is indexed by location, and the values held in it are either identifiers, integers, lexical closures, or other locations. Applying the store to a location, $\sigma(L)$, will retrieve the value or location stored there. Substitution notation, $\sigma[V/L]$, will be used to indicate changes to the store. The store returns *UNBOUND* for any location which is new; that is, has never been changed. New locations in the store are allocated using the function **new**(σ).[7] Since locations can be stored, we will use the auxiliary function **deref** to dereference locations in the store. **Deref** could be defined by:

deref(L,σ) =
    if $\sigma(L) \notin$ *Loc* **then** L
    if $\sigma(L) \in$ *Loc* **then deref**$(\sigma(L),\sigma)$

Note that **deref** always returns a location, never a value. It follows the chain of pointers in the store until it reaches a location which doesn't contain another location; this location is returned. Finally, an initial empty store will be denoted by σ_0.

Activities are the units of work for our interpreter. For Lambda there will be four different activity names: *INTERP*, *APPLY*, *+*, and *BRANCH*. The activities are records containing the

---

[7] This way of getting an unused location of the store is not entirely clean since *new*(σ) actually returns a different location each time it is called. This could be made cleaner by having *new*(σ) return both an unused location and a store, so that to allocate two locations one would do something like:

L_1, σ_1 := **new**(σ)
L_2, σ_2 := **new**(σ_1)
.....

Although more correct, this style leads to a more cluttered operational semantics later on. We hope that the use of *new*(σ) in an imperative manner is simple enough to remain clear.

activity name and one or more parameters necessary for that operation. Activities are executed by the machine, and this can result in new activities entering the activity queue, as well as updates to the store. The *INTERP* activities consist of:

1. The name *INTERP*
2. An expression in the source language, E
3. An environment, ρ
4. A destination location, L

We will notate interp activities as $\langle INTERP, E, \rho, L \rangle$. The intended effect is to evaluate expression E in the given environment and to store the answer into location L.

The *APPLY* activity will consist of:

1. The word *APPLY*
2. A rator location, L_1
3. A rand location, L_2
4. A destination location, L_3

and will be notated: $\langle APPLY, L_1, L_2, L_3 \rangle$. This activity is intended to read the rator location, L_1, and when it contains a closure, closure($\lambda x . E_1, \rho$), then the environment, ρ, is extended to map identifier x to the location of the rand, L_2. Finally, the expression E_1 is evaluated in the new environment to produce an answer which is written into location L_3.

The + activity will consist of:

1. The name +
2. A location for the first operand, L_1
3. A location for the second operand, L_2
4. A destination location, L_3

and will be notated: $\langle +, L_1, L_2, L_3 \rangle$. This activity is intended to read the two locations, L_1, and L_2, and when they are bound to integers, to store their sum into L_3.

The *BRANCH* activity will consist of:

1. The name *BRANCH*
2. A *predicate* location, L_1
3. A *consequent* expression, E_1

4. An *alternative* expression, E_2
5. An environment, ρ
6. A destination location, L_2

and will be notated: $\langle BRANCH, L_1, E_1, E_2, \rho, L_2 \rangle$. This activity implements a conditional branch by reading location L_1. When L_1 is bound, then if its value is **true** E_1 is interpreted into L_2; otherwise, E_2 is interpreted into L_2.

The *activity queue*, Act^*, is a collection of zero or more activities and is manipulated by appending or removing activities using the "•" infix operator. For example, $\langle INTERP, E_1, \rho, L_1 \rangle$•A represents an activity queue whose first element is the *INTERP* activity, and the remainder of which is denoted by A. Similarly, A•$\langle INTERP, E_1, \rho, L_1 \rangle$ denotes an activity queue whose last element is the *INTERP* activity, and whose other (preceding) elements are denoted by A. **Nil** is used to denote the empty activity queue, and **nil**•A is equivalent to A.

The interpretation of an expression in Lambda begins by creating an *INTERP* activity containing the expression along with an empty environment and the initial destination of location 0. By convention, the *answer* to a computation will be stored in location 0, so the initial state of the computation is $\langle INTERP, E, \rho_0, 0 \rangle, \sigma_0$. The machine, M, can now be described as the following state transition function:

M($\langle INTERP, E, \rho, L \rangle$•A, σ) =
  case E **of**

x	⇒ M(A, σ[ρ(x)/deref(L,σ)]) ;; case of any identifier or constant
$\lambda x.E_1$	⇒ M(A, σ[closure($\lambda x.E_1$,ρ)/deref(L,σ)])
$(E_1\ E_2)$	⇒ let L_1 := new(σ) L_2 := new(σ) M(A•$\langle INTERP, E_1, \rho, L_1 \rangle$ •$\langle INTERP, E_2, \rho, L_2 \rangle$•$\langle APPLY, L_1, L_2, L \rangle$, σ)
$(+\ E_1\ E_2)$	⇒ let L_1 := new(σ) ;; and all other binary numeric ops. L_2 := new(σ) M(A•$\langle INTERP, E_1, \rho, L_1 \rangle$ •$\langle INTERP, E_2, \rho, L_2 \rangle$•$\langle +, L_1, L_2, L \rangle$, σ)

$$(\text{if } E_1\ E_2\ E_3) \Rightarrow \text{let } L_1 := \text{new}(\sigma)\ M(A \bullet \langle INTERP, E_1, \rho, L_1\rangle \bullet \langle BRANCH, L_1, E_2, E_3, \rho, L\rangle, \sigma)$$

$$M(\langle APPLY, L_1, L_2, L_3\rangle \bullet A, \sigma) =$$
Let $L_R := \text{deref}(L_1, \sigma)$
if $\sigma(L_R) = \text{closure}(\lambda x . E, \rho)$ then $M(A \bullet \langle INTERP, E, \rho[L_2/x], L_3\rangle, \sigma)$
if $\sigma(L_R) = UNBOUND$ then $M(A \bullet \langle APPLY, L_R, L_2, L_3\rangle, \sigma)$
else *ERROR*

$$M(\langle +, L_1, L_2, L_3\rangle \bullet A, \sigma) =$$
Let $D_1 := \text{deref}(L_1, \sigma)$
$D_2 := \text{deref}(L_2, \sigma)$
if $\sigma(D_1) \in N \wedge \sigma(D_2) \in N$ then $M(A, \sigma[\sigma(D_1) + \sigma(D_2)/L_3])$
else $M(A \bullet \langle +, L_1, L_2, L_3\rangle, \sigma)$

$$M(\langle BRANCH, L_1, E_1, E_2, \rho, L\rangle \bullet A, \sigma) =$$
Let $L_P := \text{deref}(L_1, \sigma)$
if $\sigma(L_P) = UNBOUND$ then $M(A \bullet \langle BRANCH, L_1, E_1, E_2, \rho, L\rangle, \sigma)$
if $\sigma(L_P) = TRUE$ then $M(A \bullet \langle INTERP, E_1, \rho, L\rangle, \sigma)$
if $\sigma(L_P) = FALSE$ then $M(A \bullet \langle INTERP, E_2, \rho, L\rangle, \sigma)$
else error

$$M(\text{nil}, \sigma) = \text{nil}, \sigma$$

The interpreter consists of five clauses, one for each of the types of activities, and one for termination. The first clause handles the *INTERP* activities, performing a case analysis on the syntax of the expression being interpreted. If the expression is an identifier, then it is looked up in the environment, and either its associated location, or its literal value are stored in the destination location. If the expression is an abstraction, then a lexical closure is formed and stored in the destination. The interesting case is that of an application. When an *INTERP* activity for an application expression is encountered, two new locations are created in the store. One serves as a destination for the evaluation of the rator, and the other as destination for the evaluation of the rand. Two new activities are formed and enqueued into the activity queue to carry out these two evaluations in quasi-parallel. Finally, an *APPLY* activity is created and also enqueued. When additions are encountered, an + activity is created along with two *INTERP* activities to evaluate the subexpressions. Addition expressions result in the creation of *INTERP* activities for the argument expressions, and an

+ activity to add the results. Lastly, the conditional branch results in an *INTERP* activity for the predicate and a *BRANCH* activity to implement the conditional effect.

APPLY activities are interpreted by the second clause of **M**. When an *APPLY* activity is dequeued, L_1, is dereferenced. This is the location into which the rator of an application is being evaluated. The rator must evaluate into a lexical closure. If the rator location, L_R, is unbound, then this activity cannot be processed, and so the interpreter recurses after enqueueing the *APPLY* activity at the end of the queue for later processing. If the rator location contains a lexical closure, then the processing of the application can proceed. The environment is extended to map the *formal identifier* to the location L_2, which is the destination for the evaluation of the rand, and an *INTERP* activity is enqueued to evaluate the body of the closed procedure in this new environment placing the result into the destination of the *APPLY* activity. The interpretation of an *APPLY* activity does not itself affect the store. Indirectly, the body of the procedure being applied is interpreted, and it is given the destination of the *APPLY* activity to affect.

The + activities are interpreted by the third clause of **M**. When a + activity is dequeued, the operand locations, L_1 and L_2, are dereferenced. If they contain integers, then the store is updated to contain the sum at location L_3. Otherwise the activity just requeues itself. This clause of the interpreter is not really a clause but is a clause schema. It is intended to show how all primitive binary operators work, but the example is addition. All the binary operators are strict; hence, when a + or other activity is interpreted, the operand locations are dereferenced and then checked for values. If either operand is *UNBOUND*, then the activity is just requeued, otherwise the addition or other operation is done and the result is stored into the destination.

Conditional branching is handled by the *BRANCH* activity. When a conditional form (**if** E_1 E_2 E_3) is encountered in an *INTERP* activity, then a destination is set up for evaluation of the predicate, E_1, by an *INTERP* activity. A *BRANCH* activity is also enqueued to implement the decision. This *BRANCH* activity is interpreted by the fourth clause of the interpreter. The predicate location, L_1, is simply monitored for a boolean value. Depending

on the outcome of the predicate, an *INTERP* activity is enqueued to evaluate either the consequent or the alternative of the branch, E_1 or E_2. The destination for their evaluation is the location L, which is the destination of the *INTERP* activity containing the original conditional expression.

The final clause of the interpreter simply recognizes the termination condition. The termination condition for M is that the activity-queue is empty. At that point the answer is held in location 0.

2.2.2 An Example of Lambda Execution

Because of the complexity of the interpreter just shown, we will defer discussion of the correctness for a later section and proceed with an execution example. Consider evaluation of the expression `(λx.xw)λz.y`. We know from inspection that the normal form of this expression is **y**. To begin execution using the interpreter M, we start by forming an initial *INTERP* activity, using destination location 0 (zero). The initial state of the machine is then:

$\langle$*INTERP*, `(λx.xw)λz.y`, ρ_0, 0$\rangle$, σ_0

When execution begins this first *INTERP* activity is recognized by the first clause of M as an application; hence, two new locations are allocated in the store, and three new activities which are enqueued leaving the state of the machine as shown in figure 2-1.

Activity Queue

$\langle$*INTERP*, `λx.xw`, ρ_0, 1$\rangle$
$\langle$*INTERP*, `λz.y`, ρ_0, 2$\rangle$
$\langle$*APPLY*, 1, 2, 0$\rangle$

Store

0	*UNBOUND*
1	*UNBOUND*
2	*UNBOUND*

Figure 2-1: State of M after interpreting initial *INTERP* activity.

Next, the $\langle$*INTERP* `λx.xw`, ρ_0, 1$\rangle$ activity is dequeued. Since this is an abstraction, the first clause of M simply stores it into the store as a lexical closure. The same behavior occurs for the $\langle$*INTERP*, `λz.y`, ρ_0, 2$\rangle$ activity, giving the state shown in figure 2-2. At this stage, the *APPLY* activity is finally dequeued, and the second clause of M interprets it. The rator location, L_R, is location 1, which is found to contain a lexical closure of `λx.xw` and ρ_0. An

	0	*UNBOUND*
<*APPLY*, 1, 2, 0>	1	closure(λx . xw,ρ_0)
	2	closure(λz . y,ρ_0)

Figure 2-2: State of M after interpreting two *INTERP* activities containing abstractions.

extended environment is formed which maps x to the rand location, which is location 2, and an *INTERP* activity is enqueued to interpret the expression xw in this new environment, and to write the destination location 0:

<*INTERP*, xw, ρ_0[2/x], 0>

The store is left unchanged by the execution of the apply activity. This new *INTERP* activity is now the only entry in the queue, so it is dequeued and executed. Once again the expression represents an application, so two new locations are allocated in the store, which are locations 3 and 4. Two new *INTERP* activities are created, one each for the rator and the rand of the application, having as destinations locations 3 and 4 respectively. Finally, an *APPLY* activity is created leaving the state of the machine as in figure 2-3.

	0	*UNBOUND*
<*INTERP*, x, ρ_0[2/x], 3>	1	closure(λx . xw,ρ_0)
<*INTERP*, w, ρ_0[2/x], 4>	2	closure(λz . y,ρ_0)
<*APPLY*, 3, 4, 0>	3	*UNBOUND*
	4	*UNBOUND*

Figure 2-3: State of M after interpreting the first *APPLY* activity, and the following *INTERP* activity.

The next step is for the interpreter to process the <*INTERP*, x, ρ_0[2/x], 3> activity. The expression x is an identifier, so the store is updated to have ρ(x) for location 3. ρ(x) is location 2, so location 3 will now point indirectly at the contents of location 2. This illustrates why the dereferencing store is needed. Rather than wait for the values of variables to be produced, we just allocate storage cells for the values, and then copy pointers to these

cells. The next activity is now dequeued, which is <*INTERP*, w, ρ_0[2/x], 4>. In this activity, w is an identifier, but ρ(w) is w. The store is therefore updated to have w in location 4. The state of the machine after these activities is now given in figure 2-4.

<*APPLY*, 3, 4, 0>

0	*UNBOUND*
1	closure(λx.xw,ρ_0)
2	closure(λz.y,ρ_0)
3	*location 2*
4	w

Figure 2-4: State of M after interpreting two *INTERP* activities for x and w.

Now the *APPLY* activity, <*APPLY*, 3, 4, 0>, is dequeued and interpreted. The rator location is found by dereferencing location 3 to get location 2. Location 2 is found to contain a closure of λz.y, and ρ_0. An extended environment is formed, ρ_0[4/z], and an *INTERP* activity is formed using the body of the closure, y, this new environment, and the destination location 0: <*INTERP*, y, ρ_0[4/z], 0>. This activity is now the only activity so once enqueued it is immediately dequeued and recognized as an *INTERP* activity of an identifier y. The store is updated to contain ρ(y) at location 0, which is just y. Since there are no more activities, execution stops here, and location 0 contains the answer which is y as expected. The final state of the store is given in figure 2-5.

nil

0	y
1	closure(λx.xw,ρ_0)
2	closure(λz.y,ρ_0)
3	*location 2*
4	w

Figure 2-5: Final state of M after interpreting (λx.xw)λz.y.

This example has shown the method by which the quasi-parallel interpreters will execute.

Essentially, the source language is broken down syntactically into a collection of schedulable activities. The activities are kept in a FIFO queue and are repeatedly extracted from the queue and interpreted. Activities often simply requeue themselves. They can also influence the store, and can create new activities.

2.2.3 Correctness of the Quasi-Parallel interpreter for Lambda

The quasi-parallel interpreter is equivalent to the sequential interpreter, *W*, shown earlier, in that if an initial expression has a weak, head-normal form, then eventually, location 0 of the store will be updated to hold that form. Informally, we can show that the interpreter is determinate from the way the store is used. For every expression that is evaluated, a destination location is allocated in the store which is uniquely used for the value of that expression. It follows that no two activities ever have the same destination location. Since the location is freshly allocated it must contain *UNBOUND* until it is updated; hence, by the uniqueness of destinations, no location which contains a value other than *UNBOUND* is ever updated. Finally, no *APPLY*, *+*, or *BRANCH* activity ever performs an application, addition, or branch unless its required inputs have been stored. That is, these activities will wait indefinitely for values to be written into the store. They simply requeue themselves if their inputs are not available. No activity ever executes based on a location being *UNBOUND*; hence, the time when the values are stored does not matter. This makes the values stored by the interpreter independent of the order of the queueing of the activities. Determinacy of the interpreter follows since the values stored by activities always *extend* the store by changing an unbound location to a bound one, and that the order of the activities in the activity queue does not matter.

The difference between the sequential interpreter, *W*, and the quasi-parallel interpreter, **M**, arises only with respect to termination. The sequential interpreter, *W*, will terminate more often than **M**, since it is possible for an expression to create an infinite number of activities, and yet produce a normal form. An example of this is, `(λx.λy.x) 1 ((λx.xx)λx.xx)`. This form has a normal form of **1**, yet the quasi-parallel evaluation of the subexpression `(λx.xx)λx.xx` will never terminate. If we executed this expression on our parallel

interpreter, we would expect that location 0 would eventually be updated to reflect the normal form, but our termination condition that there be no more activities would never be satisfied.

2.3 The Delta Interpreter

Using a quasi-parallel interpreter like the one just shown for Lambda, we can give an interpreter for Delta. To implement Delta's primitive form of unification, we will use an auxiliary function, **bind**:

bind(Q_1, L_2, σ) =
  if $Q_1 \in Loc$ **then**
    Let D_1 := **deref**(Q_1,σ)
      D_2 := **deref**(L_2,σ)
    case
     $\sigma(D_1) = \sigma(D_2)$ **then** σ
     $\sigma(D_1) = UNBOUND$ **then** $\sigma[D_2/D_1]$
     $\sigma(D_2) = UNBOUND$ **then** $\sigma[D_1/D_2]$
     otherwise *ERROR*
  if $Q_1 \notin Loc$ **then**
    Let D_2 := **deref**(L_2,σ)
    case
     $\sigma(D_2) = UNBOUND$ **then** $\sigma[Q_1/D_2]$
     $\sigma(D_2) = Q_1$ **then** σ
     otherwise *ERROR*

Bind is similar to many unification algorithms. It takes either two locations, or a value and a location. If given two locations it "unifies" their contents, or indirects one to the other. Given a value and a location, **bind** "unifies" the contents of the location with the value. **Bind** differs from unification because it does not recursively unify any sub-terms, and also because there is no *occurs check* done to determine if a cyclic structure is formed. In the interpreter for Delta, use of **bind** to manipulate the store causes identifiers in Delta to act roughly like logical variables. Identifiers bound to locations can be affected using the == operation. An important clarification about the equality test in the **bind** definition is needed. When testing if two values are equal, we are using syntactic equality. Hence two values are equal if they are the same identifier, integer, or if they are textually identical closures.

There are several properties of the **bind** procedure that we will use later. First, **bind** never changes the value of a location other than from *UNBOUND*; once a location contains a value or a location, **bind** can only read it. Second, if two locations containing unequal values are given to **bind**, then an error occurs. The same is true if a value and a location are given to **bind**. **Bind** always returns a store which is an *extension* of the input store in that the result store always maps all bound locations to the same values, and may map some previously unbound location to a new value.

We will now present the remainder of the Delta interpreter. To eliminate excessive detail, we again interpret only the unsugared features of the language, which includes the Lambda language as presented above, plus the extended features of **(new)**, **==**, and **do**. We will also restrict the **do** form to have exactly two expressions within it: **(do** E_1 E_2**)**:

```
M(<INTERP, E, ρ, L>•A, σ) =
    case E of
     1                  ⇒ M(A, bind(E, L, σ)) ; and other integers.
     x                  ⇒ M(A, bind(ρ(x), L, σ)) ; also other identifiers.
     λx.E1              ⇒ M(A, bind(closure(λx.E1,ρ), L, σ))
     (E1 E2)            ⇒ let L1 := new(σ)
                             L2 := new(σ)
                        M(A•<INTERP, E1, ρ, L1>
                            •<INTERP, E2, ρ, L2>•<APPLY, L1, L2, L>, σ)
     (+ E1 E2)          ⇒ let L1 := new(σ) ;; and all other binary numeric ops.
                             L2 := new(σ)
                        M(A•<INTERP, E1, ρ, L1>
                            •<INTERP, E2, ρ, L2>•<+, L1, L2, L>, σ)
     (if E1 E2 E3)      ⇒ let L1 := new(σ)
                        M(A•<INTERP, E1, ρ, L1>
                            •<BRANCH, L1, E2, E3, ρ, L>, σ)
     (new)              ⇒ M(A, σ)
     (do E1 E2)         ⇒ let L1 := new(σ)
                        M(A•<INTERP, E1, ρ, L1>•<INTERP, E2, ρ, L>, σ)
     (== E1 E2)         ⇒ M(A•<INTERP, E1, ρ, L>•<INTERP, E2, ρ, L>, σ)
M(<APPLY, L1, L2, L3>•A, σ) =
    Let LR := deref(L1,σ)
    if σ(LR) = closure(λx.E,ρ) then M(A•<INTERP, E, ρ[L2/x], L3>, σ)
    if σ(LR) = UNBOUND         then M(A•<APPLY, LR, L2, L3>, σ)
                               else error
M(<+, L1, L2, L3>•A, σ) =
    Let D1 := deref(L1,σ)
        D2 := deref(L2,σ)
    if σ(D1) ≠ UNBOUND ∧ σ(D2) ≠ UNBOUND then M(A, bind(σ(D1)+σ(D2), L3, σ))
                                         else M(A•<+, L1, L2, L3>, σ)
M(<BRANCH, L1, E1, E2, ρ, L>•A, σ) =
    Let LP := deref(L1,σ)
    if σ(LP) = UNBOUND then M(A•<BRANCH, L1, E1, E2, ρ, L>, σ)
    if σ(LP) = TRUE    then M(A•<INTERP, E1, ρ, L>, σ)
    if σ(LP) = FALSE   then M(A•<INTERP, E2, ρ, L>, σ)
                       else error
M(nil, σ) = nil,σ
```

Although somewhat longer than the Lambda interpreter given earlier, the Delta interpreter is very similar in structure. There are four primary clauses to the interpreter, corresponding to the four different activities used by the machine. The first clause handles the *INTERP* activities and does a case analysis on the syntax of the expression being interpreted. The first 6 cases are nearly identical to the previous Lambda interpreter, except that the **bind** primitive is used to update the store.

The interesting cases are the language extension features, **(do** E_1 E_2**)**, **(new)**, and **(==** E_1 E_2**)**. The **(new)** expression is interpreted by doing nothing at all. The intent of **(new)** is to allocate storage, and this is achieved since anywhere that a **(new)** expression appears, a "destination" location will already have been allocated. By treating **(new)** as a no-op, we are using the destination as the allocated storage location. **(do** E_1 E_2**)** is also very simple. Its intent is to evaluate both E_1 and E_2, ignoring the value returned by E_1, and returning the value of E_2. This is achieved by simply allocating a destination for E_1, and creating two activities to interpret E_1 into the new destination, and to interpret E_2 into the original destination. **(==** E_1 E_2**)** is the binding primitive. Its intent is that E_1 and E_2 are constrained to have the same value, so that if either one of them evaluates to an unbound variable, it will take on the value of the other expression. This is achieved by creating two activities for interpreting E_1, and E_2, but using the same destination for both. The **bind** primitive then takes care of constraining the results of the two computations.

The *APPLY*, *+*, and *BRANCH* activities are interpreted in exactly the same manner as in the Lambda interpreter above.

The main result we would like to derive from this interpreter is the *determinacy* of Delta. Informally, we would like to show that for terminating programs, the contents of the store is determined only by the program, and not by the order of queueing of the activities. M as presented above is a state-transition function; hence, by its nature, it must be deterministic. However, we would like to show that Delta is deterministic even if the queueing of the activities was not handled in the FIFO manner shown; we could then conclude that Delta was a determinate language, and the determinacy was not just an artifact of a particular scheduling policy for the activities. We can conclude determinacy because of the following:

1. The **bind** primitive is the only way the store is ever affected.

2. The **bind** primitive never changes the value of a location other than from *UNBOUND*, and there is no way for any operator to test for the value *UNBOUND*.

3. In a terminating program all activities are processed. Hence, an error causing activity cannot be delayed indefinitely.

4. Conditional Branch activities wait for a value to be bound to the predicate location.

5. Apply activities wait for a value to be bound to the rator location.

6. Binary operators wait for values to be bound to both operand locations.

Points 1, 2, and 3 indicate that when the store is updated, that update is only done as a transition from *UNBOUND* to some value, and there are never two activities racing to update a location with different values; this situation will always cause an error because all such activities must execute in order for the interpreter to terminate. Points 3, 4, and 5 just show that the only action that any activity takes based on a location containing *UNBOUND*, is to requeue the activity for later processing. In other words the activities wait for values to appear in locations; they do not race by checking for a location to be empty at a given time. Consequently, the conditional branch activity evaluates only one of the two branches based on the value stored into the predicate location, and not on when that value is stored. It follows that the order of the queueing of activities does not matter, since the order of their scheduling cannot affect the value stored in any location or the outcome of a conditional branch. Since the order of the queueing does not matter, and the store is only updated in the extensional fashion of the bind primitive, we can conclude that Delta is determinate and that the interpreter, **M**, does not introduce any indeterminacy.

2.3.1 An Example of Delta Execution

Finally, to clarify the workings of the Delta interpreter we will show an example execution of an inherently non-sequential code fragment similar to one given earlier:

```
((λx.
    (+ (* x x)
       (do (== x 8)
           (- x 5)))) (new))   ; should result in 67
```

This example is rather contrived, and serves only to illustrate the execution of the interpreter, M. It is not intended to demonstrate a proper programming methodology for use of `(new)` and `==`. The example creates an unbound variable, `x`, using the `(new)` feature. It then uses `x` in an arithmetic expression and in a `do` form. The `==` operation will affect the value of the variable `x` so that the entire expression takes on a value of 67.

The initial state of the machine is:

$\langle INTERP, ((\lambda x.(+ (* x x) (do (== x 8) (- x 5)))) (new)), \rho_0, 0\rangle, \sigma_0$

Step 1 of the execution is to look at this activity and to recognize that it is an application. Two new locations (1 and 2) are allocated for the rator and rand of the application, and three activities are produced leaving the processor in the state shown in figure 2-6.

$\langle INTERP,$
$(\lambda x.(+ (* x x)$
$(do (== x 8)$
$(- x 5)))), \rho_0, 1\rangle$
$\langle INTERP, (new), \rho_0, 2\rangle$
$\langle APPLY, 1, 2, 0\rangle$

0	*UNBOUND*
1	*UNBOUND*
2	*UNBOUND*

Figure 2-6: State of Delta Interpreter after Step 1.

Step 2 dequeues the next activity which is an *INTERP* activity of the lambda abstraction. This is executed by storing a closure into destination location 1. The next activity is an *INTERP* activity of the expression `(new)` so the activity is simply discarded. Step 3 is to dequeue the *APPLY* activity. The rator location is location 1, which contains a closure, so a new environment is formed which maps the identifier `x` onto location 2. The body expression, `(+ (* x x) (do (== x 8) (- x 5)))` is enqueued as part of an *INTERP* activity using this new environment, and the destination location 0. Since this is the only activity, it is immediately dequeued and found to be a binary addition. Two new locations are allocated (locations 3 and 4) and three activities are generated resulting in the state shown in figure 2-7.

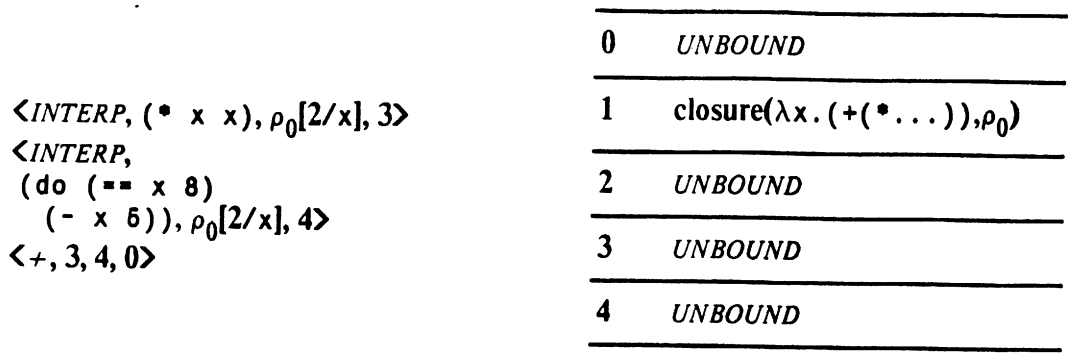

Figure 2-7: State of Delta Interpreter after Steps 2 and 3.

Step 4 is to dequeue the activity involving the `(* x x)` expression. This is also found to be a binary operator, so two more locations are allocated (locations 5 and 6) and three more activities are added to the queue. Next the *INTERP* activity involving the expression `(do (== x 8) (- x 5))` is dequeued, and found to be a `do` expression. One additional location is allocated (location 7) and two activities are added to the queue resulting in the state of figure 2-8.

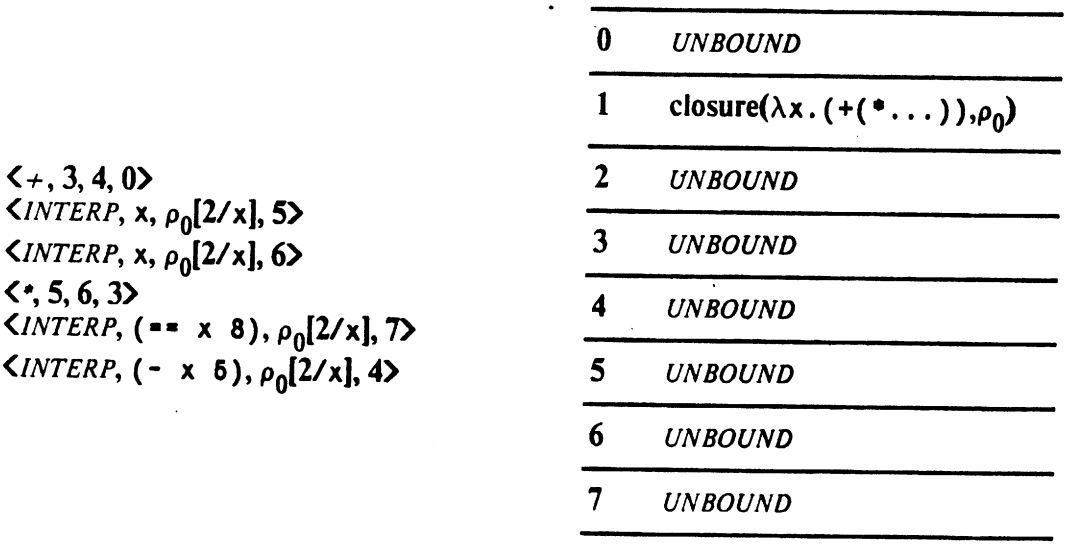

Figure 2-8: State of Delta Interpreter after Step 4.

Step 5 dequeues an + activity, but since its operand locations, 3 and 4, are not yet bound to values, it is simply requeued and the next activity dequeued. This activity is <*INTERP*, x, ρ_0[2/x], 5> which executes by binding the destination location 5, to the location

associated with identifier x (location 2). The next activity, <*INTERP*, x, ρ_0[2/x], 6>, is processed similarly and results in the state given in figure 2-9.

<*, 5, 6, 3>
<*INTERP*, (== x 8), ρ_0[2/x], 7>
<*INTERP*, (- x 5), ρ_0[2/x], 4>
<+, 3, 4, 0>

0	*UNBOUND*
1	closure(λx.(+(*...)),ρ_0)
2	*UNBOUND*
3	*UNBOUND*
4	*UNBOUND*
5	*location 2*
6	*location 2*
7	*UNBOUND*

Figure 2-9: State of Delta Interpreter after Step 5.

Step 6 is to dequeue the <*, 5, 6, 3> activity. Dereferencing location 5 gives location 2 which is still unbound, so this activity is simply requeued. The next activity dequeued is <*INTERP*, (== x 8), ρ_0[2/x], 7>. This activity is an equate operation, so two new activities are enqueued leaving the machine state as in figure 2-10.

<*INTERP*, (- x 5), ρ_0[2/x], 4>
<+, 3, 4, 0>
<*, 5, 6, 3>
<*INTERP*, x, ρ_0[2/x], 7>
<*INTERP*, 8, ρ_0[2/x], 7>

0	*UNBOUND*
1	closure(λx.(+(*...)),ρ_0)
2	*UNBOUND*
3	*UNBOUND*
4	*UNBOUND*
5	*location 2*
6	*location 2*
7	*UNBOUND*

Figure 2-10: State of Delta Interpreter after Step 6.

Step 7 dequeues the next activity which is <*INTERP*, (- x 6), ρ_0[2/x], 4>. This activity represents a binary subtraction, so two new locations are allocated (locations 8 and 9) and three activities are enqueued. The next activity is the <+,...> which will simply be requeued as will the <*,...> following it in the queue. The next interesting state of the machine is shown in figure 2-11.

<*INTERP*, x, ρ_0[2/x], 7>
<*INTERP*, 8, ρ_0[2/x], 7>
<*INTERP*, x, ρ_0[2/x], 8>
<*INTERP*, 6, ρ_0[2/x], 9>
<-, 8, 9, 4>
<+, 3, 4, 0>
<*, 5, 6, 3>

0	*UNBOUND*
1	closure(λx.(+(*...)),ρ_0)
2	*UNBOUND*
3	*UNBOUND*
4	*UNBOUND*
5	*location 2*
6	*location 2*
7	*UNBOUND*
8	*UNBOUND*
9	*UNBOUND*

Figure 2-11: State of Delta Interpreter after Step 7.

Step 8 dequeues <*INTERP*, x, ρ_0[2/x], 7>. This results in binding locations 2 and 7 in the store, which means that location 7 contains an indirection to location 2. The next activity is <*INTERP*, 8, ρ_0[2/x], 7>, and when it is interpreted, the bind primitive is called with the value 8 and the location 7. Location 7 is dereferenced giving location 2 where the 8 is stored. This is the crucial step in the interpretation of this expression. The binding of the value 8 with location 7 ends up storing the 8 in location 2, which is where all the other operators are expecting to find the value of identifier x. The use of binding and dereferencing here is allowing the non-local effect of the := operator to propagate back to the original location given to the unbound variable x. Our state is now given in figure 2-12.

In step 9, the next two activities work similarly to those of step 8. The first activity dequeued

<$\langle$*INTERP*, x, ρ_0[2/x], 8$\rangle$
<$\langle$*INTERP*, 5, ρ_0[2/x], 9$\rangle$
<$\langle$-, 8, 9, 4$\rangle$
<$\langle$+, 3, 4, 0$\rangle$
<$\langle$*, 5, 6, 3$\rangle$

0	*UNBOUND*
1	closure(λx.(+(*...)),ρ_0)
2	8
3	*UNBOUND*
4	*UNBOUND*
5	*location 2*
6	*location 2*
7	*location 2*
8	*UNBOUND*
9	*UNBOUND*

Figure 2-12: State of Delta Interpreter after Step 8.

associates location 2 and location 8, and the second stores the value 5 into location 9 leaving the state in figure 2-13.

Step 10: the <-, 8, 9, 4> activity has resurfaced. Dereferencing location 8, we get location 2 which is bound. Location 9 is also bound so the result of the subtraction, 3, is calculated and the destination is updated using the bind primitive. The next activity is <+, 3, 4, 0> but it will simply be requeued because location 3 is not yet bound to a value. The <*, 5, 6, 3> activity is dequeued next. This time locations 5 and 6 both dereference to location 2, which is bound to the value 8. The product, 64, is stored in the destination location 3 using the bind primitive. Once this is done, the only remaining activity is <+, 3, 4, 0> which will now be interpretable since locations 3 and 4 now contain values. The sum, 67, is written into the destination which is location 0. Since there are no more activities at this point, execution terminates resulting in the final state given in figure 2-14.

In summary, the queueing and dequeueing of activities allows the interpreter to simulate a parallel execution by essentially *time-sharing* among the different subsections of the original expression. The use of the **bind** primitive and the dereferencing of locations in the store

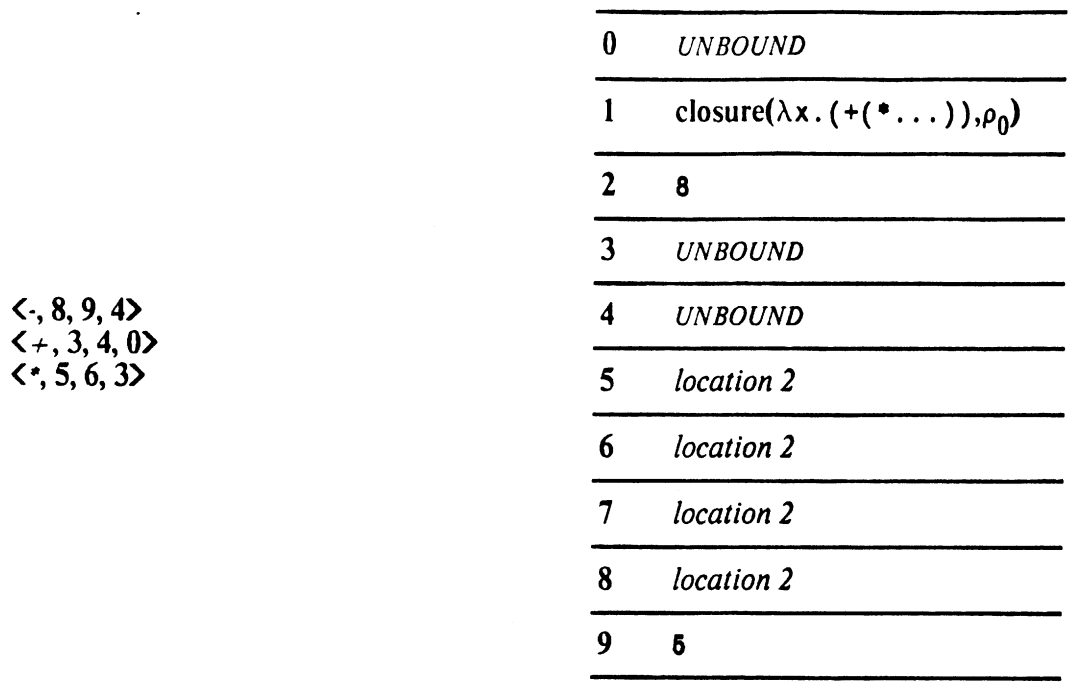

Figure 2-13: State of Delta Interpreter after Step 9.

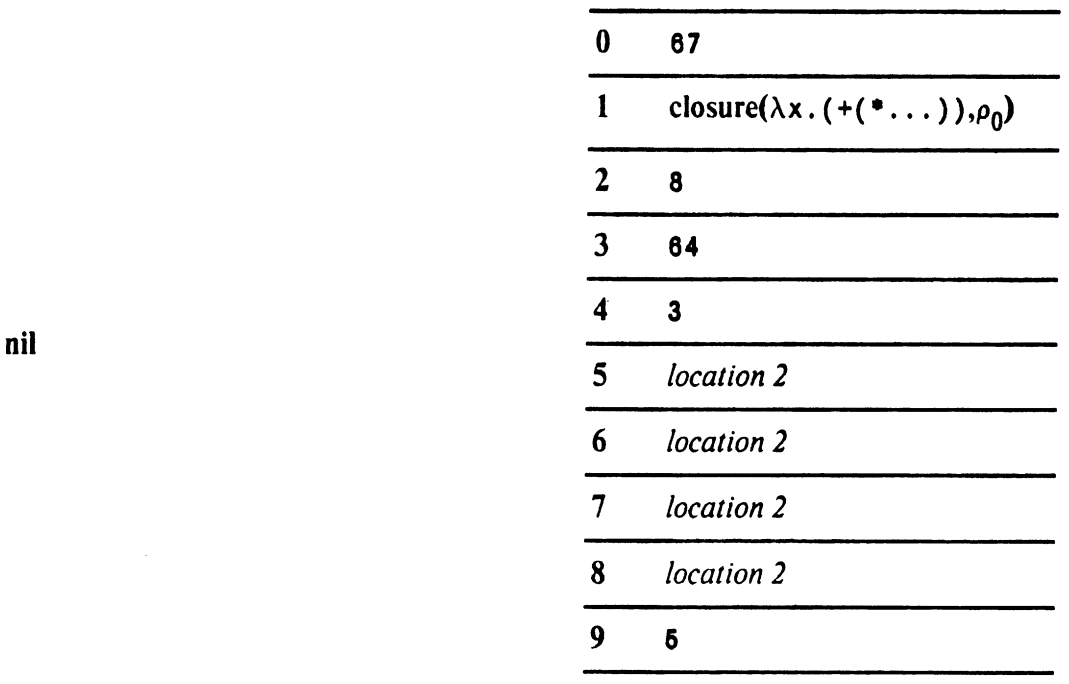

Figure 2-14: Final State of Delta Interpreter

allows the constraint placed by the (== E_1 E_2) operation to propagate to the unbound variables involved.

2.4 Programming in Delta

As in Lambda, our functional language, Delta is devoid of any data-structuring features. However, Using the *new* feature and the tuple technique shown previously for Lambda, we can easily construct a form which *allocates* a specific length tuple of unbound variables:

```
(let ((allocate-4-tuple (λ(n)
        (tuple
          (new) (new) (new) (new))))) ;make a four tuple of unbound variables
  ......)
```

Conceptually, it is very easy to generalize this technique so that the form **(allocate n)** returns a tuple of length *n* of unbound variables. As for Lambda, we will assume this extension exists, along with the function **select** described earlier. **replace** is not needed as a built-in function in Delta, since it can be written within the language. Intuitively, using **select** on a tuple of unbound variables should select out one of the variables in such a way that using == on it will affect the original tuple. For example:

```
(let ((x (allocate 2)))      ; allocate a 2 tuple
  (let ((z (select 1 x))     ; z is first element
        (w (select 2 x)))    ; w is second element
    (do (== z 5)             ; equate z and 5. This affects x.
        (== w 7)             ; equate z and 7. This affects x too.
        x)))                 ; return the updated tuple.

  ;; the result should be the tuple 5,7.
```

This effect is achieved in our interpreter for variables in closures because of the **bind** primitive and the use of dereferencing, so we will assume that this same behavior appears for allocated structures.

The crucial difference between Lambda and Delta should now be apparent. **(allocate n)** produces an object which can be distributed to several parts of the program for production or consumption of the variables in it. Tuples in Lambda must be produced all at once and can only be distributed for consumption.

2.5 Flat Structures in Delta

At this point we can look at the additional expressive power that Delta has over Lambda by writing and analyzing the two test programs: inverse-permute, and tree-append. The ability to create unbound variables and constrain them later allows a much more efficient and natural version of inverse-permute:

```
(letrec
   ((iterate-loop (λ(index A B)
                    (if (> index (tuple-length A)) nil
                        (do (== (select index B) (select (select index A) A))
                            (iterate-loop (+ index 1) A B)))))
;------------

    (inverse-permute (λ(A)
                       (let ((B (allocate (tuple-length A))))
                         (do (iterate-loop 0 A B)
                             B)))))
;------------
  (inverse-permute (tuple 2 3 5 4 1)))  ;;; perform the algorithm.
;;; the answer is the tuple 3,5,1,4,2.
```

This program now resembles an imperative implementation using assignments much more than the functional version since almost every aspect of it is using the non-local binding effect of the `==` operation. The `inverse-permute` routine simply allocates an array of unbound variables of the appropriate size, and then calls `iterate-loop` to fill them in with appropriate values from the array `A`. The inner `iterate-loop` procedure actually performs O(n) non-local binding operations, so this program requires only O(n) time and space. However, the program is not performing assignments as it would if it were written in Fortran, since we only assert equality constraints on the variables.

In summary, the loss of referential transparency caused by introducing `(new)` and `==` into a functional-style language results in a language with a useful form of non-local effect which allows flat structures to be manipulated much more efficiently. The language remains determinate.

2.6 Deep Append in Delta

Although it is not immediately apparent, `allocate` and `==` will not allow us to write a better program for the tree append problem. Unfortunately, there are certain programs which still cannot be expressed in Delta, namely those in which the essence of the algorithm is to check to see if a location is unused, and if so, to acquire and exploit that location. To understand this limitation, let us try to write the program in an *imperative* Lisp language. This language will be syntactically exactly like our functional language, Lambda, but with added assignment operators. Once again assume that we have a tuple constructor called

make-node which makes a node of a tree containing a **left-subtree**, **right-subtree**, and **node-value**, where the corresponding field of a node is selected using a function with the same name. The fields will also be assigned using the forms: **set-left-subtree**, **set-right-subtree**, and **set-node-value**. We will again use **nil** to represent the empty tree, and the empty list. An efficient tree append program will need O(n) tree nodes to represent the tree, and will not perform any copying of the tree during its construction. Appending each integer to the tree requires O(log n) operations on average or O(n) operations in the worst case. The best we can hope for then is O(n log n) time and O(n) space for tree append. The imperative version of the program is then:

```
(letrec
;--------
   ((append-integer (λ(int tree)

        ;; appends an integer to an existing tree.

        (if  (null? tree)   ;; the first case.. tree is empty

                (make-node nil nil int)

              (do
                 (if (< int (node-value tree))  ;; else compare to current node value

                     (set-left-subtree tree     ;; update left subtree
                             (append-integer int (left-subtree tree)))


                     (set-right-subtree tree    ;; update right subtree
                             (append-integer int (right-subtree tree)))

               tree)))))       ;; return the tree as the answer.
;--------

    (tree-append (λ (list-of-ints tree)

        ;; appends elements of list one at a time.

        (if (null? list-of-ints) tree   ; done, so return the finished tree.

            (tree-append (cdr list-of-ints)   ;; append one and recurse
                  (append-integer (car list-of-ints) tree))))))
;---------

 (tree-append (list 4 3 5 2 6) nil))  ;now try it out.
```

Only the **append-integer** routine is different from the program as written in the Lambda language. The **append-integer** routine tests **tree** to see if it is null, and if so it has reached a leaf of the tree, so it creates a node containing the integer and returns it. If **tree** is not null, then it must be a tree node, so the procedure compares the value at that node of

the tree with the integer, `int` to determine which subtree it should be appended to. A recursive call to **`append-integer`** will return a tree node, and the appropriate field of the current tree node (the value of the variable **`tree`** is replaced with the returned node. For example, if the tree is currently only a single root node with both left and right subtrees null, then appending another integer will simply replace one of these null subtrees with a newly created tree node. Clearly, this program exhibits the storage efficiency we want. It allocates only enough nodes to hold all the tree elements, and updates the pointer structure to assemble the tree. It achieves this by reusing the storage locations of the tree nodes. Originally they hold `nil`, to signify empty subtrees, but they are later updated to contain new subtrees. As we mentioned in the introduction, the assignment statements and reusable store of sequential programming languages make it difficult or impossible to expose parallelism in programs. On the other hand, we would like to achieve this same level of storage efficiency in our parallel programming language.

Unlike an imperative language which reuses storage locations, the Delta language can allocate new locations as unbound variables and later define them only once. Because of this no additional dependencies are introduced, since there is no reuse of locations.[8] The behavior of the tree nodes in the imperative program above is to start with value `nil`, and then change once into tree nodes. This parallels to the way logical variables work, which begin as *UNBOUND*, and later take on values. As a result, it is attractive to attempt a Delta program for the deep append program which uses the same algorithm as the imperative version, which is essentially this: *Build the tree so that the leaves of the tree are always unbound variables. When appending to a leaf, simply equate an existing unbound leaf to be a new node containing its own new unbound leaves.* This strategy leads to a program for appending a new integer into the tree which is something like:

[8] Real implementations of these languages would rely on garbage collectors to reclaim storage when it is no longer accessible. A survey of garbage collection techniques is found in [12]

```
(letrec
;--------
   ((append-integer (λ(int tree)

        ;; appends an integer to an existing tree.

        (if  the tree is an unbound variable    ;; check if this is unbound -- a leaf.

              (do
                 (== tree (make-node (new) (new) int)) ;; add the new node at the leaf
                 tree)                                 ;; return the tree as answer

              (do
                 (if (< int (node-value tree))  ;; else compare to current node value

                     (append-integer int (left-subtree  tree))  ;append to left
                     (append-integer int (right-subtree tree))) ;or to right for effect

               tree))))         ;; return the tree as the answer.

     .....)
```

The reason this program doesn't work is that we need to check if the tree is an unbound variable to decide whether we have reached a leaf and can now append a new node. If the tree is bound, then it must be another tree node, so we must descend recursively. There is no test in the Delta language which allows us to check if a variable is unbound. All the constructs in Delta except == require that variables are bound. The constructs of Delta allow us to create unbound variables, to constrain them, and to equate them by use of the == operator, but there is no way to tell if an expression represents a bound or unbound variable.

Logic programming languages can express the deep-append problem efficiently, and still avoid assignment statements. If we look at the Prolog [8] version of this program, we can see how this is achieved:

```
tree-append([], Tree).

tree-append([Int|R], Tree) :-
       append-integer(Int, Tree)
     & tree-append(R, Tree)
     & close-tree(Tree).

append-integer(Int, node(Left, Right, Int)).

append-integer(Int, node(Left, Right, Value)) :-
       Int < Value
     & append-integer(Int, Left).

append-integer(Int, node(Left, Right, Value)) :-
       Int >= Value
     & append-integer(Int, Right).

close-tree([]).
```

```
close-tree(node(Left, Right, Value)) :-
        close-tree(Left)
    & close-tree(Right).
?:- tree-append([4, 3, 5, 2, 6], Tree).
```

The program consists of three definitions: **tree-append**, **append-integer**, and **close-tree**. **Append-integer** is the important part of the tree append program. The first clause succeeds at appending the integer to the tree if the tree is an unbound logical variable. In that case, the tree is unified with a node which contains the appended integer and two unbound variables **Left** and **Right** which are the subtrees of the node. If the tree is already bound to a node containing a different integer then the first clause fails and the second clause is tried. The second and third clauses of **append-integer** handle the case of recursing down the left and right branches of the tree respectively. The interesting behavior here is that the program essentially tests to see if the tree is an unbound variable, and if it is, it exploits that fact immediately by unifying the variable with a new node and succeeding in the first clause. If the tree is not an unbound variable, then the unification fails and the other clauses are tried. The unification process either succeeds by exploiting an unbound variable, or fails indicating that that variable was already in use. This is tantamount to having a test for *UNBOUND* which can be used to give a result for a conditional branch; however, it combines it neatly as an atomic operation with a binding of the variable. Prolog can attempt unifications conditionally, and the backtracking mechanism allows it to behave in different ways depending on the success or failure of the attempts. **Close-tree** is used after the appending of all the integers is complete; it recursively descends the tree unifying all the unbound "ends" with **nil**. This is done by attempting to unify each tree node with **nil**, and branching to the second clause of **close-tree** to recurse when the unification fails. Note that **close-tree** is done only after the appending of all the integers has completed, according to a Prolog-style execution order.

Looking back at the semantics for Delta, we see that the == operator, which implements Delta's trivial sort of unification, does not behave in a conditional fashion. (== E_1 E_2) equates E_1 and E_2 by giving them the same destination location; hence, they are "unified" by the **bind** primitive. If they do not "unify" in this way, a run-time error occurs. In conclusion, the small amount of "logical" behavior that variables in Delta have does not

provide Delta with all the expressive power of a logic programming language. Our next extended language, *Eta*, will allow us to exploit some of this conditional behavior of unification within the operational framework we have already set up. It will not embody the automatic backtracking or unification of Prolog, but will glean enough of the conditional binding behavior to be able to solve the deep-append problem efficiently.

2.7 Input and Output from Delta Programs

By adding logical variables to a functional language we seem to gain the expressive power of streams for performing input and output [41, 4]. In the language IC-Prolog [11], streams are just lists containing logical variables. Delta will also be able to implement streams in this way.

The built in function `input` can be assumed to yield a list of all inputs from the user's terminal as typed a line at a time. The list is made up of cells which are actually made by an `(allocate 2)` form evaluated in Delta. The built in function `output` will perform just the opposite. It will take as argument a list of lines to be printed on the output device. The pointer structure of a list will be used to constrain the order of actual input or output events.

The key observation is the following. As one attempts to read deeper and deeper into the input list, one cannot read farther than the part that has been defined by the actual input system, and hence the program's operators will wait for the physical inputs to occur. Output is symmetric to this. The output system cannot read deeper into the output stream than the user's program has defined. The ability to leave an unbound variable at the "tail" of the output list allows output to be done in a very natural almost imperative fashion.

For example, let us assume we want to write a program to simply echo the lines typed to the input:

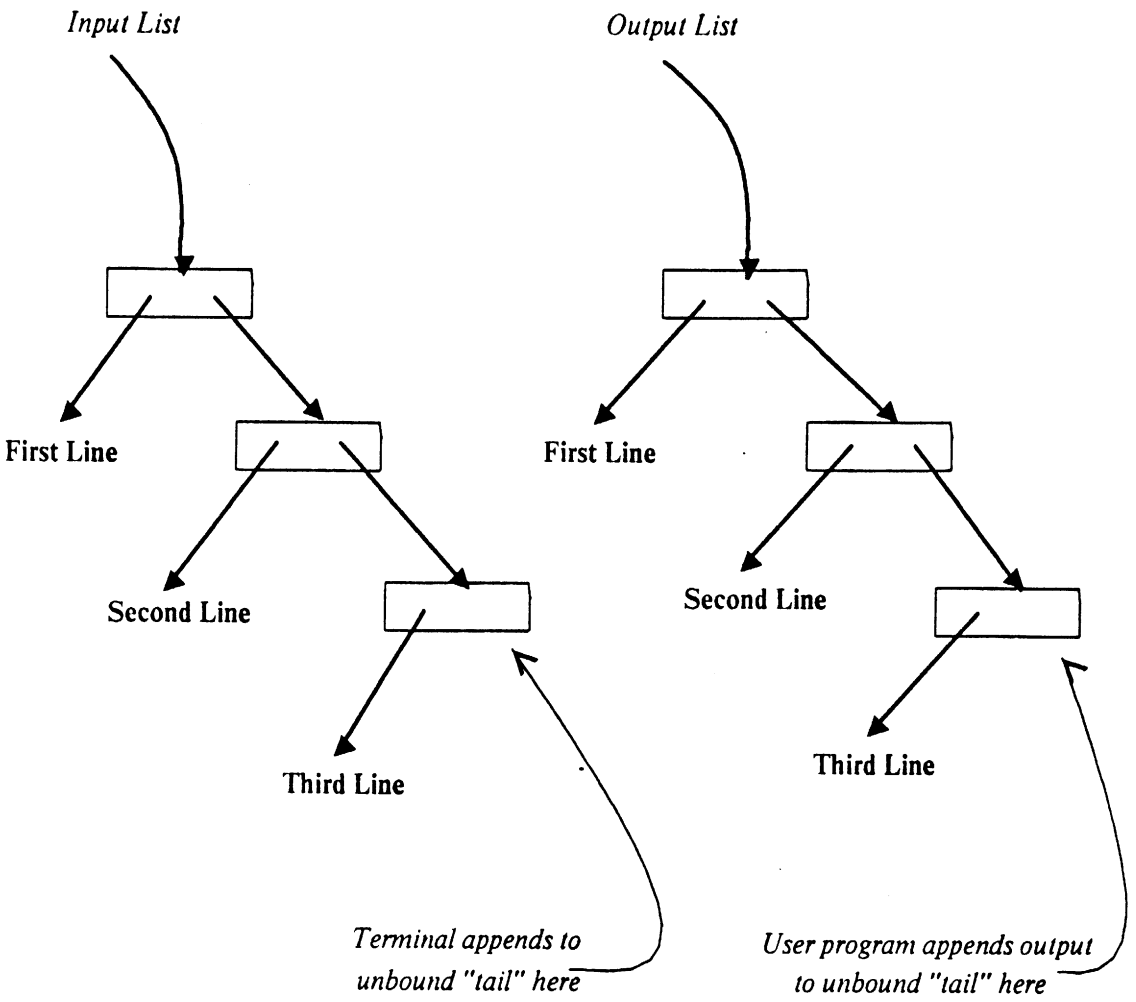

Figure 2-15: Terminal input/output on streams with "unbound tails".

```
(letrec ((echo-loop (λ(input-list output-list)
           (do
            (== output-list (allocate 2))        ;; new cell for output list
            (let ((first-line-in (car input-list))
                  (rest-input    (cdr input-list))
                  (first-line-out (car output-list))
                  (rest-output    (cdr output-list)))
                  (do
                     (== first-line-out first-line-in)        ;; echo the line.
                     (echo-loop rest-input rest-output)))))))  ;; repeat

  (let ((input-list (input))
        (output-list (new)))   ;; initially unbound since we're producing it.
    (do
      (output output-list)     ;; give it the unbound and let it wait until
                               ;; we get around to defining it.
      (echo-loop input-list output-list))))
```

The program consists of only one routine, **echo-loop**, and a main body. The main body first calls the **input** procedure and binds the variable **input-list** to the result. Successive car's of this list will be successive inputs. The variable **output-list** is introduced as a unbound variable, and the built-in procedure **output** is called on it. This will output successive car's of **output-list** in order. **Echo-loop** simply takes the input and output lists, and by defining the unbound variables it defines successive car's of **output-list** to be **==**, that is, equated to the successive car's of **input-list**. The behavior of this program is shown in figure 2-15.

This program has a considerably simpler representation in the IC-Prolog language:

```
?- input(X) & output(X).
```

In IC-Prolog, variables can stand for streams. In this example, the variable **X** stands for the entire history of all input. By providing **X** as an argument to **output**, the program is ensuring that the output is the same as the input. In Delta, this technique for I/O can be generalized into a general communication technique between sections of a program. One part can produce a list while another section reads it simply by sharing a variable which represents the list. Since the choice of sections of the program doing this may be input dependent, a dynamically evolving network of communication can result. The implications of this for programming methodology are beyond the scope of this thesis.

Chapter Three

Conditional Binding of Logical Variables

3.1 The Eta Language

We have seen that Delta has some advantages in expressive power over the pure functional language Lambda. Now we will undertake a further extension to the language, and we will call the further extended language *Eta*. The Delta language has some of the properties of logical variables; however, it has no notion of success and failure of an equate operation. We would like the ability to attempt to constrain a variable to have a particular value, and to branch conditionally based on the success or failure of the try.

The additional feature of the new Eta language is the `=?=` operator. `=?=` is similar to `==` in Delta, except that it does not cause an error if its two argument expressions are not "unifiable". Rather, it returns true or false depending on whether such an operation succeeds or fails. `=?=` will be able to have a non-local effect if its first argument is unbound, by binding this first variable to the value of the second, and returning `true`. Suppose x and y are unbound variables created using the `(new)` operation, then:

`(=?= x 1)` ⇒ `true` and `x` gets the value 1,
`(=?= 1 2)` ⇒ `false`,
`(=?= x y)` can't be reduced.

Note that the `=?=` operator is not symmetric like `==` was in Delta; it has a definite left-to-right behavior.[9] `=?=` should be thought of as performing a *conditional binding*. The binding available in Delta did not have this conditional behavior, and could be called *absolute binding*. The `==` operation can be thought of as a clean way to do what operationally amounts to a single assignment for effect. The distinction between absolute binding, and conditional binding is important since indeterminacy is introduced into the language. For example, the following program has an indeterminate result:

[9] This is necessitated by the details of the operational semantics given later.

```
(let ((x (new)))
    (do
      (=?= x 20)
      (=?= x 30)
      x))
```

This program returns either 20, or 30 as the answer depending on which **=?=** operation is performed first, but it does not make use of the boolean values returned by the **=?=** operations. In order to program using **=?=** we will also need a construct: **(after** E_1 E_2**)** which is much like the **do** construct in Delta. Both E_1 and E_2 are evaluated, and the value of the expression is the value of E_2. However, the **after** construct causes the forms to be evaluated in sequence; that is, E_2 is not evaluated until E_1 has returned a value.

It is important to realize that Eta is a much simpler language than a real Logic Programming language in some respects. It does not include full unification or *don't-know* non-determinism. For example, a Prolog program which is not as easily expressed in Eta is:

```
member(X, [X|Y]).
member(X, [Z|Y]):- member(X,Y).
sum-10(X,Y,Set):- member(X,Set), member(Y,Set), sum(X,Y,10).
?- sum-10(A, B, [1, 3, 5, 4, 6]).
```

This program consists of two definitions. The first is the standard Prolog definition of **member** which determines if an element is a member of a list. The second is called **sum-10**, which given a set represented as a list, produces two numbers X, and Y, which are in the list, and whose sum is 10. This program uses a common Prolog programming paradigm, *generate and test.* The two calls to the **member** predicate in the definition of **sum-10** are generators of members of the set, and the test is the call to the predicate **sum**. Prolog uses automatic backtracking to enumerate the members of the sets, and will ultimately try all pairs of elements to see if their sum is 10. To write this program in Eta, one would have to write a generate and test procedure, essentially implementing what is built into the Prolog interpreter.

On the other hand, since Eta allows higher-order procedures, it has some expressive power that is not present in current Logic-based languages, which are all essentially first-order. For example, it is possible to write the reduce function:

```
(letrec ((reduce (lambda (f)
                   (lambda (list)
                     (if (null? (cdr list)) (car list)
                         (f (car list) ((reduce f) (cdr list))))))))
  (let ((sum-list (reduce +)))
    (sum-list '(1 3 6 7)))) ;; answer is 16
```

Reduce is a higher-order function. It takes a binary operator as its first argument, and returns a function which given a list of values, *inserts* that operator between the values of the list. In the example, **reduce** is used to build **sum-list**, which adds up all the elements of a list. The elegance and power of this programming style have been described in [35, 19]. Some higher-order features can be incorporated into Prolog [40], but their status with respect to the foundations of logic programming is questionable.

3.2 Operational Semantics for Eta

To understand exactly the additional power that the **=?=** operator gives us, we will next present a quasi-parallel interpreter for Eta. The interpreter is much like the Delta interpreter; however, there are important differences. The language is no longer deterministic, so we must make some accommodation for this in our interpreter. The Delta interpreter broke the program down into activities, which were placed into the activity-queue, a *FIFO*. For Eta, we will allow any activity to be selected from the queue by making the **M** interpreter non-deterministic. We will do this by adding an extra rule for **M**: $\mathbf{M(F \cdot A, \sigma) = M(A \cdot F, \sigma)}$. This rule is applicable any time there is more than one activity in the queue, and it allows the queue to be shuffled by moving an activity to the rear without looking at it. To insure that programs terminate, we will still require that every activity is eventually selected for interpretation by **M**; that is, we do not leave any activity unprocessed for an unbounded amount of time.

Most of the interpreter is identical to that given for Delta and is omitted; only the additional clauses are shown here:

M(⟨*INTERP*, E; ρ, L⟩•A, σ) =
  case E of

    the first 8 cases are the same as for Delta

    (▪?▪ E_1 E_2) ⇒ let L_1 := new(σ)
                       L_2 := new(σ)
                       M(A•⟨*INTERP*, E_1, ρ, L_1⟩•⟨*INTERP*, E_2, ρ, L_1⟩•⟨=?=, L_1, L_2, L⟩, σ)

    (after E_1 E_2) ⇒ let L_1 := new(σ)
                         M(A•⟨*INTERP*, E_1, ρ, L_1⟩•⟨*WAIT*, L_1, E_2, ρ, L⟩, σ)

M(⟨=?=, L_1, L_2, L_3⟩•A, σ) =
  Let D_1 := deref(L_1, σ)
      D_2 := deref(L_2, σ)
  if $\sigma(D_2)$ = *UNBOUND* then M(A•⟨=?=, L_1, L_2, L_3⟩, σ)
  if $\sigma(D_1)$ = *UNBOUND* then M(A, σ[L_2/L_1][true/deref(L_3, σ)])
  if $\sigma(D_1)$ = $\sigma(D_2)$ then M(A, σ[true/deref(L_3, σ)])
                                    else M(A, σ[false/deref(L_3, σ)])

M(⟨*WAIT*, L_1, E, ρ, L_2⟩•A, σ) =
  if σ(deref(L_1, σ)) = *UNBOUND* then M(A•⟨*WAIT*, L_1, E, ρ, L_2⟩, σ)
                                                 else M(A•⟨*INTERP*, E, ρ, L_2⟩, σ)

M(F•A, σ) = M(A•F, σ)

The Eta interpreter has two new types of activities: a *=?=* activity, and a *WAIT* activity. These are created by the first clause of M when the syntactic forms (▪?▪ E_1 E_2) and (after E_1 E_2) are encountered in *INTERP* activities.

The *=?=* activity has three locations associated with it. L_1 is assumed to evaluate into an unbound variable; that is, it is assumed to end up referencing an unbound location. L_2 is assumed to eventually receive a value, and the activity waits for it to become bound to a value, simply requeueing itself if L_2 is unbound. When L_2 takes on a value, then an attempt to bind it with L_1 is made. If this attempt is successful, then L_3 receives **true**, otherwise it gets **false**. Non-determinism appears in the interpreter since several of these *=?=* activities can exist in the queue simultaneously. If several of these are *enabled*, that is, they have their respective L_2 locations bound to values, and if they share a common

location for L_1 which is unbound, then whichever of the activities is dequeued first will cause **true** to be stored in its destination L_3. All the others will store **false** in their destinations. Essentially, several *=?=* activities can race to bind a common L_1 location. Whichever is scheduled first will succeed and store **true**; the others will fail and store **false**. The last clause of the interpreter, **M(F•A, σ) = M(A•F, σ)**, introduces real non-determinism in the scheduling of activities.

The *WAIT* activity implements the semantics of the **after** construct. It keeps an expression, environment, destination, and also a *trigger* location. When the trigger location becomes bound, then the *WAIT* activity simply enqueues an *INTERP* activity to interpret the expression into the destination. If the trigger location is unbound, then the *WAIT* activity just requeues itself to be retried later.

3.2.1 An Example of Eta Execution

To see the non-determinism that this interpreter exhibits, we can interpret the expression presented earlier. Once again, this example is too trivial to show any programming methodology, but is sufficient to illustrate the operational capabilities of Eta:

```
(let ((x (new)))
    (do
      (=?= x 20)
      (=?= x 30)
      x))
```

This expression can be "desugared" so that our interpreter will execute it directly:

```
((λx.(do (=?= x 20) (do (=?= x 30) x))) (new))
```

The initial state of the interpreter would then be:

<*INTERP*, ((λx.(do (=?= x 20) (do (=?= x 30) x))) (new)), ρ_0, 0>, σ_0

Evaluation of this initial activity will lead to allocation of two new locations (1 and 2), and the creation of three other activities:

<*INTERP*, (λx.(do (=?= x 20) (do (=?= x 30) x))), ρ_0, 1>
<*INTERP*, (new), ρ_0, 2>
<*APPLY*, 1, 2, 0>

The first two activities will execute when they are selected from the queue, and will result in the storing of a closure of the lambda expression into location 1. Interpreting the *APPLY*

activity will result in creating an extended environment, $\rho_0[2/x]$, and an activity for evaluating the body of the expression:

<*INTERP*, (do (=?= x 20) (do (=?= x 30) x)), $\rho_0[2/x]$, 0>

This is the only activity at this point, so it is dequeued and interpreted. Since it is a **do** form, an additional location is allocated (location 3), and two activities are enqueued:

<*INTERP*, (=?= x 20), $\rho_0[2/x]$, 3>
<*INTERP*, (do (=?= x 30) x), $\rho_0[2/x]$, 0>

Thus far, the only operation that has affected the store was the creation of the lexical closure which was stored into location 1. Since our interpreter now allows reordering of the activity queue, let us next select the *INTERP* activity of the **do** expression. Once again a new location is allocated (location 4), and two new activities are generated, leaving the queue of activities as:

<*INTERP*, (=?= x 20), $\rho_0[2/x]$, 3>
<*INTERP*, (=?= x 30), $\rho_0[2/x]$, 4>
<*INTERP*, x, $\rho_0[2/x]$, 0>

Selecting next the activity involving the expression (=?= x 20), an interpreter clause specific to the Eta language is now used. Two new locations, (5 and 6) are allocated, and three new activities are created:

<*INTERP*, x, $\rho_0[2/x]$, 5>
<*INTERP*, 20, $\rho_0[2/x]$, 6>
<=?=, 5, 6, 3>

Similarly, when the activity involving (=?= x 30) is selected, then two more locations are allocated (7 and 8), and three more activities are created leaving the activity queue with:

<*INTERP*, x, $\rho_0[2/x]$, 5>
<*INTERP*, 20, $\rho_0[2/x]$, 6>
<=?=, 5, 6, 3>
<*INTERP*, x, $\rho_0[2/x]$, 7>
<*INTERP*, 30, $\rho_0[2/x]$, 8>
<=?=, 7, 8, 4>
<*INTERP*, x, $\rho_0[2/x]$, 0>

Because of the non-determinism in our interpreter, and for clarity, we will next select the activities which evaluate constants. The activity

<*INTERP*, 20, $\rho_0[2/x]$, 6>

will simply result in the value **20** being stored in location 6 by the **bind** primitive. Similarly, the activity involving the **30** will result in the value **30** being stored in location 8. This leaves the state of the interpreter as shown in figure 3-1.

<*INTERP*, x, ρ_0[2/x], 5>
<*=?=*, 5, 6, 3>
<*INTERP*, x, ρ_0[2/x], 7>
<*=?=*, 7, 8, 4>
<*INTERP*, x, ρ_0[2/x], 0>

0	*UNBOUND*
1	**closure(λx.(do...),ρ_0)**
2	*UNBOUND*
3	*UNBOUND*
4	*UNBOUND*
5	*UNBOUND*
6	**20**
7	*UNBOUND*
8	**30**

Figure 3-1: State of Eta interpreter after storing constants **20** and **30**.

We will next select the activities which interpret the expression **x**, since they influence only the store. In each case, the activity results in indirection of the destination location to the location associated with identifier **x** which is location 2. Hence locations 5, 7, and 0 will all end up containing references to location 2. The state of the interpreter is then as show in figure 3-2.

At this point, the only two activities left are both *=?=* activities. Furthermore, both are *enabled* in that whichever one is selected next for execution will in fact execute. Non-deterministically let us choose the **<*=?=*, 7, 8, 4>** activity for execution. Looking back at the interpreter clause for *=?=* activities, we dereference locations 7 and 8 to get locations 2 and 8. Since location 8 is bound to the value **30**. we update the store so that location 2 refers to location 8. By dereferencing. location 2 now refers to the value **30**. We also update location 4 to contain the value **true**. This leaves the state as in figure 3-3.

	0	*location 2*
	1	closure(λx.(do...),ρ_0)
	2	*UNBOUND*
	3	*UNBOUND*
<=?=, 5, 6, 3> <=?=, 7, 8, 4>	4	*UNBOUND*
	5	*location 2*
	6	20
	7	*location 2*
	8	30

Figure 3-2: State of Eta interpreter before executing =?= activities.

	0	*location 2*
	1	closure(λx.(do...),ρ_0)
	2	*location 8*
	3	*UNBOUND*
<=?=, 5, 6, 3>	4	true
	5	*location 2*
	6	20
	7	*location 2*
	8	30

Figure 3-3: State of Eta interpreter after executing first =?= activity.

Finally, the second =?= activity is executed. Locations 5 and 6 are dereferenced to give locations 8 and 6. Since location 6 is bound to the value 20 and location 8 is bound to the value 30, the activity compares these values and stores false in the destination location 3. At this point execution terminates since there are no more activities. The final state is given in figure 3-4.

	0	*location 2*
	1	closure(λx.(do...),ρ_0)
	2	*location 8*
	3	`false`
nil	4	`true`
	5	*location 2*
	6	**20**
	7	*location 2*
	8	**30**

Figure 3-4: Final state of Eta interpreter.

If we had selected the =?= activities for execution in the other order, the final state would differ since location 2 would have been bound to location 6 instead of location 8. Dereferencing location 0 gives location 8 which contains the answer value of **30**.

This example is quite trivial, and does not make use of the `after` feature of Eta for controlling the non-determinism. It should be clear, however, where the non-determinism is introduced into Eta programs. The next section will exhibit more elaborate Eta programs, which are too large to analyze at the level of detail just shown, but they will illustrate practical ways to use the non-determinism and conditional binding effect of the =?= operation.

3.3 Deep Append in Eta

The absolute binding effect of == operations in Delta allowed us to efficiently solve the flat structure problem, but not to adequately solve the deep append problem. We will now look at how the further extended capabilities of *Eta* provide a solution to deep appending. As in Delta, we will continue to use the same model of data structures as tuples which can contain unbound variables. The addition of the =?= operation to Delta will allow us to write a

better tree-append program, which works more like the Prolog example given earlier. As in our original tree-append program in Lambda, we will use a tree node tuple built with the constructor **make-node**. This constructs a vertex of the tree having a **left-subtree**, a **right-subtree**, and a **node-value** which are selected using functions of the same names. **nil** will represent the empty subtree or the empty list. The regular list operations of **car**, **cdr**, and **list** are also used.

```
(letrec
;--------
    ((tree-append (λ (list-of-ints tree)
        (if (null? list-of-ints) tree   ; done, so return the finished tree.
            (after
              (append-integer (car list-of-ints) tree)  ; append one integer
             (after                                     ; then
              (tree-append (cdr list-of-ints) tree)     ; append the rest
             ;; then close off the tree.                ; then
              (close-tree tree))))))                    ; close off the tree
;--------
    (append-integer (λ (int tree)
        (if (=?= tree (make-node (new) (new) int)) nil       ; done
              (if (< int (node-value tree))             ;; compare to tree root
                  (append-integer int (left-subtree tree))   ; put it into left
                  (append-integer int (right-subtree tree))) ; put it into right
              )))
;---------
    (close-tree (λ (tree)
                 (if (=?= tree nil) nil                 ; done
                     (do
                       (close-tree (left-subtree tree))       ; close left
                       (close-tree (right-subtree tree)))))))) ; close right
;---------
  (let ((tree (new)))              ; create an unbound variable as the tree.
    (do   (tree-append (list 4 3 5 2 6) tree)  ; append the list
          tree)))                                ; return the tree
```

Like the Prolog version given earlier, this program is broken into three sections: **tree-append**, **append-integer**, and **close-tree**. The key feature of this new deep-append program is the use of the **=?=** test in the **append-integer** routine. **=?=** is used to test if the tree is equal to the new item that we want to append to the tree. Because of the ability of **append-integer** to leave unbound values in the records it appends, the program can build the tree from the root downward, never having to copy tree nodes as the functional version did, so the storage requirement is only O(n). Unlike the Prolog version,

however, the *Eta* program must enforce its own sequentialities. The **after** constructs used in **tree-append** are needed to insure that the tree is built in the proper order, and that the tree is completed before the **close-tree** routine goes about trying to "seal up" the unbound variables contained in the tree.

Using **=?=** in the above program has reduced the copying overhead, but in making the program storage efficient we have also made it sequential by use of the **after** construct. It is interesting to look at a different version of this program. Let us introduce a variation on the **after** construct: **(after2** E_1 E_2 E_3**)**. **After2** will work similarly to **after**. The value of the expression is the value of E_3; however, instead of waiting for the evaluation of just a single expression we will evaluate E_1 and E_2 in quasi-parallel and wait for both to store a value in their destinations. It is easy to augment the state transition machine **M** to handle this operation:

M($\langle$*INTERP*, E, ρ, $L$$\rangle$•A, σ**) =**
  case E **of**

  (after2 E_1 E_2 E_3**)** ⇒ **let** L_1 **:= new(**σ**)**
    L_2 **:= new(**σ**)**
    M(A•$\langle$*INTERP*, E_1, ρ, $L_1$$\rangle$
      •$\langle$*INTERP*, E_2, ρ, $L_2$$\rangle$
      •$\langle$*WAIT2*, L_1, L_2, E_3, ρ, $L$$\rangle$, σ**)**

M($\langle$*WAIT2*, L_1, L_2, E, ρ, $L_3$$\rangle$•A, σ**) =**
  if σ**(deref(**L_1, σ**))** = *UNBOUND* ∨ σ**(deref(**L_2, σ**))** = *UNBOUND*
  then M(A•$\langle$*WAIT2*, L_1, L_2, E, ρ, $L_3$$\rangle$, σ**)**
  else M(A•$\langle$*INTERP*, E, ρ, $L_3$$\rangle$, σ**)**

The first clause here shows the decomposition of the form **(after2** E_1 E_2 E_3**)** into three activities. The first two are to interpret the subexpressions E_1 and E_2 in quasi-parallel. The third is to wait for both of these to store their values into their respective locations, and then to evaluate E_3. This is done by means of the new *WAIT2* activity. The interpretation of *WAIT2* activities is given in the second clause. We can now change the definition of **tree-append** to use the **after2** construct.

```
(letrec
;--------
    ((tree-append (λ (list-of-ints tree)
        (if (null? list-of-ints) tree   ; done, so return the finished tree.
            (after2
              (append-integer (car list-of-ints) tree) ; append one integer
              (tree-append    (cdr list-of-ints) tree) ; append the rest
              ;; then close off the tree.             ; then
               (close-tree tree))))))                  ; close off the tree
    ......)
```

This new version will exhibit very different behavior. Instead of appending the integers to the tree in sequence, the recursive call to **tree-append** will unfold creating a potentially large number of concurrently active versions of **append-integer**. These will all be attempting to bind the "root" of the tree simultaneously, but according to the semantics of **=?=** only one of them will succeed. The remaining active *processes* will race to bind the subtrees of the root, and so on. The actual tree which is produced will be a binary search tree containing all the elements of the input list; however, it will have been built in a non-deterministic order. The program now has more parallelism since many of the concurrently active processes can be overlapped as they compare with integers that have already been appended to the tree, but this extra parallelism has also made the program indeterminate. As an answer we get one of a number of possible trees of integers. It seems that the indeterminacy here is of a controllable sort, since we may not care which binary search tree is ultimately produced. The **after2** construct is still needed to delay the closing of the tree until after all the integers have been appended to it. Although this extension has allowed us to build a tree non-deterministically, it is still not providing us with the kind of non-determinism which is implemented in Prolog through backtracking. Eta's non-determinism is of the "don't-care" variety, in that our example program builds one of a set of possible trees and we don't care which one is produced. This is unrelated to the non-determinism of Prolog, which is commonly called "don't-know" non-determinism since it implies search for a solution.

3.4 Programming with Non-determinism

Extensions have been proposed to functional languages to allow programming using non-determinacy for systems applications [4]. The basis of the mechanism is to add a non-deterministic merge operator to the language, along with suitable constructs to enforce a reasonable discipline in use of the construct.

Similar applications can be programmed directly in Eta, since a non-deterministic merge can be written in the language:

```
(letrec ((nmerge (λ (x y) ;; x and y are streams, that is lists to be merged.
          (let ((x1 (car x))
                (xr (cdr x))
                (y1 (car y))
                (yr (cdr y))
                (first (new))
                (rest (new)))
        (do
            (if (=?= first x1) (== rest (nmerge xr y)) nil)
            (if (=?= first y1) (== rest (nmerge x yr)) nil)
            (cons first rest))))))
  (nmerge (list 1 1 1) (list 2 2 2)))
```

Nmerge takes two lists, and creates a third, whose elements are all the elements in the input lists, interleaved in an arbitrary manner. In conclusion, the Eta language is certainly as expressive as a functional language extended with non-deterministic merge.

Chapter Four

Conclusions

4.1 Summary

The goal of this thesis has been to show that the functional programming style can be enhanced with features from logic programming languages, and that the resulting languages are more powerful for manipulating data structures. Starting from the functional language, Lambda, the enhancement can be done in a two stage process. Adding the ability to create an unbound variable and to constrain it later with the equate operation gave us the Delta language. Delta allowed us to manipulate arrays more easily and provided a means of doing I/O; unfortunately, Delta is not referentially transparent which makes equivalence of programs more difficult to determine. This certainly impacts the ease of program transformation negatively. Delta is, however, a determinate language. Extending the capabilities of the language further gave us Eta. Eta contains a conditional form of the equate operation allowing one to try to equate two expressions, and to branch conditionally based on success or failure. Eta is as capable as Prolog at solving the deep append problem, but in our quasi-parallel execution model Eta is not determinate. Using lists, Eta can simulate the non-deterministic merge features proposed as extensions to functional languages. Although it is not proven here, this thesis provides some evidence that deep-append and similar programming problems cannot be solved efficiently without introducing non-determinism into the programming language.

The languages described here form the lower part of a hierarchy of expressiveness in languages. Functional languages are the least expressive, followed by Delta-class, followed by Eta-Class. Realistic languages with efficient parallel execution models can be developed based on Delta. An example of this is the ID language [27]. ID is a functional language extended with I-structures, giving it similar expressive power to Delta. Moreover, ID is designed to be executed on the Tagged Token Dataflow Architecture [2].

The additional =?= feature of Eta does not complicate the language significantly, and provides the expressive power of non-determinism to the language. This is needed for systems programming on parallel machines. Operationally, the conditional equate operation should be only slightly more complex than a regular == operation. Also, since both Delta and Eta have a functional language as a subset, any implementation technique useful for functional programs can be applied to the functional subset of Delta or Eta.

A still higher level of the hierarchy contains languages with goal-directedness, or automatic backtracking on logical variables, as well as the higher-order abstractions possible in functional languages. To our knowledge, research combining Logic and Functional programming in this way has restricted the language to have only first-order functions [17, 28, 16, 32, 31].

There are some issues left unresolved by the previous discussions of the extended languages. These include cyclic objects, run-time errors, and demand-driven evaluation.

4.1.1 Cyclic Data Structures

It is possible in Delta or Eta, to produce cyclic data structures. For example, the following program creates a *cons-cell* whose car and cdr both refer back to itself:

```
(let ((x (new))
      (y (new)))
  (let ((c (cons x y)))
    (do (== x c)   ;; this forms the car cycle
        (== y c)   ;; this forms the cdr cycle
        c)))       ;; return the cell.
```

This implies that these languages cannot rely on simple storage reclamation strategies such as reference counting. Functional programming advocates have generally discounted the utility of having cyclic data structures, since one can have an acyclic data structure such as an edge list, which represents a cyclic graph. This adds one level of interpretation to any use of cyclic objects, and there are applications where cyclic data is useful. Rather than over play this issue, it is enough to say that there seem to be many applications which can use cyclic structures profitably, such as: network databases, dataflow graph compilers, type checkers and type inference systems, lisp interpreters, semantic networks, circuit simulators, etc.

4.1.2 Run Time Errors

Use of the `(allocate n)` feature of Delta introduces new possibilities for run time errors into the functional framework. Programs may deadlock, if the variables that they attempt to read are never defined, or they may "overconstrain" a location, that is, write it twice. Neither of these situations was possible with simple tuples in the functional language. The new error situations are actually reasonable because they are quite analogous to *bounds checking* errors. Bounds checking must be done at run time, since an arbitrary computation may be used to generate an index into a tuple. Since Delta does not allow one to write a location twice, if a program does so, then there is an error in the program. Probably what was intended was to write some other location, but a bug led to the accidental generation of the same subscript for more than one "equate" operation. Generating an illegal index is a reasonable run time error. Deadlock could be caused also by an indexing error; that is, attempting to read the wrong location, or never writing one. In any case, the errors do not seem that unreasonable.

4.1.3 Demand Driven Evaluation

Throughout this thesis, we have used only a data-driven notion of parallel execution. Demand driven execution is an equally viable technique for the execution of functional programs, and it provides the ability to manipulate "infinite" data objects.

Unfortunately the non-sequential nature of our languages implies that demand-driven evaluation is inherently difficult. If a program in Delta produces unbound variables, and the value of one of them is needed, there is no way to know what computation to start up in order to produce the value. One could of course start up any computation that could possibly influence the variable, but that is not in the spirit of true demand driven execution, since excess work would be done to compute elements of the structure which are not actually needed. The problem of demanding the value of an unbound variable is analogous to the problem of demanding the solution to a logical or of two boolean expressions. One need only evaluate either of the two expressions to true to produce the demanded value. To demand the value of an unbound variable, one would have to demand all computations

which could result in any constraint on that variable. However one only really needs the single computation which ultimately gives that variable its value. Because of these difficulties, we avoided presenting a demand-driven model for any of the languages here. Lindstrom [25] has described a subset of a functional language extended with logical variables using a complex variation of demand driven evaluation, but he does not deal with the issue of indexable data structures composed of these variables or with dynamic procedure invocation in these languages.

4.2 Comparison to Related Work

There has been a significant amount of research on integrating Logic programming and Functional programming. The key and unique aspect of this thesis is the restriction of unification to the behavior of equate. This simplifies the operational semantics of the languages to the extent that abstract interpreters could be presented directly. Equate also is a great deal less complicated to implement than true unification based binding. Our approach has been far more operational than most because of the issues that arise in Delta programs that have no adequate sequential semantics, and because of the desire to compare the expressive power of the languages in a reasonable framework. The focus of most other research has been on integrating Logic and Functional Programming in a harmonious manner retaining as much of both paradigms as possible.

Most of the work on combining Logic and Functional programming evolves from the idea of adding additional equality axioms to the rules making up a logic program. These equality rules can be used by an extended unification algorithm to rewrite terms in a manner much like reduction for functional languages [32, 31, 22] Several researchers are now pursuing a mechanism called *narrowing*, which is a generalization of term rewriting and resolution [28, 16, 17]. Reddy [29] has written an article which tries to clarify the relationship between Logical and Functional programming. These languages all differ from the approach of this thesis in that they retain the non-determinism of Logic programming, and add only first-order functions to the framework. Only Lindstrom [25] has looked at adding logical variables to functional languages within a fully deterministic framework.

4.3 Directions for Future Research

There are at least two areas of future research which are immediately related to this thesis. First, the ability to exploit parallel processors for searching in parallel is an area that has not been addressed by functional programs. Search problems are common in artificial intelligence programs, and the very notion of searching implies that some "wasted" work must be done. In a parallel machine where there is excess processing capacity, one would like to have an easily controlled way of using many processors so that parallel threads of computation each search in their own part of the search space possibly interacting with the other branches for rapid pruning. Logic programming languages with backtracking seem to be able to expose this *OR-parallelism*, but do not seem to provide a framework in which it can easily be controlled. The approach of enhancing the function-style framework with an explicit search capability may be easier to implement for practical systems.

Second the programming methodology used in languages with logical variables is an important consideration. Use of logical variables allows arbitrary communication between sections of a program through data structures. Reasonable conventions for how this technique should be used are needed to avoid writing programs with opaque structure.

References

1. W.B. Ackerman. A Structure Processing Facility for Dataflow Computers. Proceedings of the 1978 International Conference on Parallel Processing, July, 1978.

2. Arvind, D.E. Culler, R.A. Iannucci, V. Kathail, K. Pingali, and R.E. Thomas. "The Tagged Token Dataflow Architecture". Laboratory for Computer Science, MIT, Cambridge, Mass., July, 1983. (Prepared for MIT Subject 6.83s).

3. Arvind, V. Katail, and K. Pingali. Sharing of Computation in Functional Language Implementations. 1984 International Workshop on High Level Computer Architecture, University of Maryland, Dept. of Computer Science, College Park, Maryland., 1984.

4. Arvind and J. D. Brock. "Resource Managers in Functional Programming". *Journal of Parallel and Distributed Computing 1*, 1 (January 1984).

5. J. Backus. From Function Level Semantics to Program Transformation and Optimization. Research Report RJ 4567, I.B.M. Research, 1985.

6. L. Bic. Execution of Logic Programs on a Dataflow Architecture. Dept. of Information and Computer Science, Univ. of California, Irvine., November, 1983.

7. R. Book. *Formal Language Theory: Perspectives and Open Problems.* Academic Press, New York, 1980.

8. D. L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, D. H. D. Warren. DECsystem-10 PROLOG User's Manual. Department of Artificial Intelligence, University of Edinburgh, November, 1982.

9. L. Cardelli. ML under Unix. Bell Laboratories, Murray Hill, N.J., 1983.

10. A. Ciepielewski and S. Haridi. A formal Model for OR-Parallel Execution of Logic Programs. Proceedings of the IFIP 1983, 1983.

11. K. Clark and F.G. McCabe and S. Gregory. IC-PROLOG - Language Features. In *Logic Programming*, K.L. Clark and S.-A. Tarnlund, Ed., Academic Press, London, England, 1982.

12. J. Cohen. "Garbage collection of linked data structures". *ACM Computing Surveys 13*, 3 (September 1981).

13. J.S. Conery, and D.F. Kibler. Parallel Interpretation of Logic Programs. Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981.

14. J. Darlington and M. 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, 1981, pp. 65-76.

15. J.B. Dennis. "Data Flow Supercomputers". *Computer 13*, 11 (November 1980), 48-56.

16. N. Dershowitz and D.A. Plaisted. Logic Programming cum Applicative Programming. Proceedings of the International Symposium on Logic Programming, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, June, 1985.

17. J.A. Gocuen and J. Meseguer. "Equality, Types, Modules, and (Why not?) Generics for Logic Programming". *Journal of Logic Programming 1*, 2 (August 1984).

18. S. Haridi and A. Ciepielewski. An OR-Parallel Token Machine. TRITA-CS-8303, Royal Insitiute of Technology, Dept of Telecommunication Systems-Computer Systems, Stockholm, Sweden., 1983.

19. P. Henderson. *Functional Programming: Application and Implementation.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

20. R.M. Keller, G. Lindstorm, and S. Patil. A Loosely-Coupled Applicative Multi-Processing System. AFIPS Conference Proceedings, June, 1979, pp. 613-622.

21. J.W. Klop. *Mathematical Centre Tracts.* Volume 127: *Combinatory Reduction Systems.* Mathematisch Centrum, Amsterdam, 1980.

22. W. Kornfeld. Equality for Prolog. Proceedings of the 8th International Joint Conference on Artificial Intelligence, IJCAI, 1983.

23. R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*,
North Holland, 1974, pp. 556-574.

24. D.J. Kuck, R.H. Kuhn, D.A. Padua , B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. Proceeding of ACM Symposium on Principles of Programming Languages, January, 1981.

25. G. Lindstrom. Functional Programming and the Logical Variable. Proceedings of the 12th ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, New York, 10036, January, 1985.

26. G.A. Mago. A Cellular Computer Architecture for Functional Programming. COMPCON Spring 80, February, 1980, pp. 179-187.

27. R. Nikhil and Arvind. Id Nouveau. Laboratory For Computer Science, MIT, July, 1985. (Prepared for MIT Subject 6.83s).

28. U. Reddy. Narrowing as the Operational Semantics of Functional Languages. Proceedings of the International Symposium on Logic Programming, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, June, 1985.

29. U. Reddy. On the Relationship Between Logic and Functional Languages. Draft version.

30. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* M.I.T. Press, 1977.

31. P. A. Subrahmayam. Pattern Driven Lazy Reduction: A Unifying Evaluation Mechanism for Functional and Logic Programs.

32. Subrahmayam and You. Funlog = Functions + Logic: A Computational Model Integrating Functional and Logic Programming. Proceedings of the 1984 International Symposium on Logic Programming, IEEE Computer Society, February, 1984.

33. Tanaka, Amamiya, Tanaka, Kadowaki, Yamamoto, Shimada, Sohma, Takizawa, Ito, Takeuchi, Kitsuregawa, Goto. The Preliminary Research of Data Flow Machine and Data Base Machine as the Basic Architecture of Fifth Generation Computer Systems. Proceedings of the 1st International Conference on 5th Generation Computer Systems, Institute for New Generation Computing Technology, Tokyo, 1982.

34. K.R. Traub. "An Abstract Parallel Graph Reduction Machine". In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE Computer Society, Boston, 1985, pp. 333-341.

35. D. A. Turner. The Semantic Elegance of Applicative Languages. Functional Programming Languages and Computer Architecture, October, 1981, pp. 85-92.

36. D.A. Turner. Miranda: A non-strict functional language with polymorphic types. Proceedings of the Conference on Functional Programming Languages and Computer Architecture, 1985.

37. Uchida, Tanaka, Tokoro, Takei, Sugimoto, Yashuhara. New Architectures for Inference Mechanisms. Proceedings of the 1st International Conference on 5th Generation Computer Systems, Institute for New Generation Computing Technology, Tokyo, 1982.

38. S. Umeyama and K. Tamura. A Parallel Execution Model of Logic Programs. Conference Proceedings of the 10th Annual International Symposium on Computer Architecture, 1983.

39. P. Wadler. Listlessness is Better than Laziness. 1984 ACM Symposium on Lisp and Functional Programming, Association for Computing Machinery, New York, New York, 10036, 1984.

40. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence*, J.E. Hayes, D. Michie, and Y-H Pao, Ed., Ellis Horwood Ltd., West Sussex, England, 1982, pp. 441-454.

41. K.-S. Weng. Stream-Oriented Computation in Recursive Data Flow Schemas. TM-68, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1975.

42. C. Zaniolo. Object-Oriented Programming in Prolog. Proceedings of the International Symposium on Logic Programming, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, February, 1984.

CS-TR Scanning Project
Document Control Form

Date : 3/30/95

Report # LCS-TR-356

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 20(80-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page (2)
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGEMAP	(1-2) UNNUMBERED TITLE AND ABSTRACT PAGES
	(3-70) PAGES #'ED 3-70
	(71-77) SCANCONTROL, COVERA, COVERB, SPINE, PRINTERS NOTES, DOD(2)
	(78-80) TRGTS

Scanning Agent Signoff:

Date Received: 3/30/95 Date Scanned: 4/7/95 Date Returned: 4/7/95

Scanning Agent Signature: Michael W. Cook

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-356	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Logical Structures for Functional Languages		5. TYPE OF REPORT & PERIOD COVERED M.S. Thesis June 1984-Feb. '86 6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-356
7. AUTHOR*(s)* Michael J. Beckerle		8. CONTRACT OR GRANT NUMBER*(s)* DARPA/DOD N000-14-83-K-0125
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE February 1986 13. NUMBER OF PAGES 70
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public Release, distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Functional Programming, Logic Programming, Interpreters, Parallelism

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Functional Programming is frequently advocated as an appropriate programming discipline for parallel processing because of the difficulty of extracting parallelism from programs written in conventional sequential programming languages. Unfortunately, the use of Functional operations often implies excessive copying or unnecessary sequentiality in the access and construction of data structures. Logic Programming languages can use logical variables to manipulate data structures more easily; however, parallel implementations of them are not well understood.

(cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Two new programming languages which extend Functional languages with some of the additional expressive power of logical variables for manipulation of data structures are introduced. These new languages are studied in the context of two programs which cannot be expressed efficiently in a Functional language: the flat-structure problem, and the deep-append problem. The first new language allows the flat-structure problem to be solved efficiently, but loses the referential transparency of Functional languages. The second allows the deep-append problem to be solved also, but loses the property of determinacy.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/07/1995

M.I.T. Libraries
Document Services