MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 311 August 1988

Analyzing the State Behavior of Programs

Alan Bawden

Abstract

It is generally agreed that the unrestricted use of state can make a program hard to understand, hard to compile, and hard to execute, and that these difficulties increase in the presence of parallel hardware. This problem has led some to suggest that constructs that allow state should be banished from programming languages. But state is also a very useful phenomenon: some tasks are extremely difficult to accomplish without it, and sometimes the most perspicuous expression of an algorithm is one that makes use of state. Instead of outlawing state, we should be trying to understand it, so that we can make better use of it.

I propose a way of modeling systems in which the phenomenon of state occurs. I propose that systems that exhibit state-like behavior are those systems that must rely on their own nonlocal structure in order to function correctly, and I make this notion of nonlocal structure precise. This characterization offers some new insights into why state seems to cause the problems that it does. I propose to construct a compiler that takes advantage of these insights to achieve some of the benefits normally associated with purely functional programming systems.

1 Introduction

As functional programming languages and parallel computing hardware become more widespread, understanding the phenomenon of *state* has becoming increasingly important for computer science. It is generally agreed that the unrestricted use of state can make a program hard to understand, compile, and execute, and that these problems increase in the presence of parallel hardware.

The usual approach to controlling these problems is to impose programming language restrictions on the use of state, perhaps even by ruling it out altogether. Others have proposed schemes that accept state as a necessity, and try to minimize its bad effects. [16, 2, 1, 6, 10]

I would like to propose that before either outlawing state, or learning to tolerate it, we should first try to better understand it in the hope of eventually being able to reform it. This paper takes some steps towards such an understanding by proposing a way of modeling systems in which the phenomenon of state occurs. Using this model we will be able to characterize those systems in which some components of a system perceive other components as having state. We will gain some insight into why state is such a problem. Finally I will suggest how this insight might lead us towards better ways of thinking about state, and make our programming languages more expressive when we program with state. I will also propose the construction of a compiler that uses these insights into the nature of state to achieve some of the benefits normally associated with purely functional programming systems.

1.1 What is State?

It's not immediately clear what, if anything, the word "state" refers to. We ordinarily treat State as being a property of some *object*. We pretend that state can be localized in certain portions of the systems we construct. We act as if the question "where is the state?" has an answer. Ordinarily this doesn't get us into any trouble, but if we try to analyze systems from a global perspective, this view becomes untenable.

It cannot be the case that state is an attribute possessed by an object independent of its observer. In a system consisting of an observer and some other components, in which the observer describes one component as having state, it is often possible to provide an alternate description in which some other component contains the state. Often the system can be redescribed from a viewpoint in which another component is treated as the observer and the original observer appears to be the component with state. Sometimes the system can even be described in such a way as to eliminate all mention of state. (In [15] Steele and Sussman explore this mystifying aspect of state in some depth.)

In cases where state cannot be eliminated, it behaves much like a bump in a rug that won't go away. Flatten the bump out in one place, and some other part of the rug bulges up. Any part of the rug can be made locally flat, but some global property (perhaps the rug is too large for the room) makes it impossible for the entire rug to be flat simultaneously. Analogously, we may be able to describe all the components of a system in stateless terms, but when the components are assembled together, some components will perceive other components as possessing state.

As an example, consider the simple system consisting of a programmer interacting, via a keyboard and display, with a computer. Imagine that the software running on the computer is written entirely in a functional programming language, the stream of output sent to the display is expressed as a function of the stream of keyboard input. (See [9] for a demonstration of how this can be done.) Thus the description of the subsystem consisting of the keyboard, computer and display is entirely free of any mention of state, yet from the programmer's viewpoint, as he edits a file, the computer certainly *appears* to have state.

Imagine further that the programmer is actually a robot programmed in a functional language, his stream of keystrokes is expressed as a function of the stream of images he sees. Now the situation appears symmetrical with respect to programmer and computer, and the computer can claim that that it is the programmer that is the component of the system that has state.

All components in this system agree that from their perspective there is state somewhere else in the system, but since each component is itself described in state-free terms, there is no component that can be identified as the location of that state. This does *not* mean that the phenomenon of state is any less real than it would be if we could assign it a location. It does mean that we have to be careful about treating state as anything other than a perceptual phenomenon experienced by some components in their interaction with other components. In particular, we must not expect to single out components as the repositories of state.

Therefore an important aspect of my approach to studying state will be a reliance on observers *embedded in the system itself* to report on state as they experience it. A more conventional approach would be to treat state as something experienced by observers *external* to the system under study. Mine is a much more minimalist approach, demanding less of state as a phenomenon. State is certainly experienced by entities within the systems that we construct, but this does not imply that state can be studied as if it were a property *of* those entities.

This is similar to the stand taken by those physicists who advocate the Many Worlds interpretation of quantum mechanics [5], and I adopt it for similar reasons. By dispensing with external acts of observation, and instead treating observation solely as a special case of interaction between the components of a system, the Many Worlds formulation gives insight into why observers *perceive* effects such as the Einstein-Podolsky-Rosen "paradox".

The programs we write are really instructions to be followed by little physicists who inhabit computational universes that we create for them. These embedded observers must react to their environment on the basis of their perceptions of it. They are not privy to the god's-eye view that we, as the creators and debuggers of their universe, are given.

Since programming languages are designed to facilitate the instruction of these little physicists, it is natural that programming languages describe phenomena as they are perceived by such embedded observers, but that does not mean that we should adopt the same terminology when we study the universe as a whole. The notion of *state* is a valid one, in as much as it describes the way one component of a system can appear to behave to another, but it would be a mistake to conclude from this that state is a intrinsic property that we, as external investigators, can meaningfully assign to certain components.

By carefully restricting the notion of state to apply only relative to embedded observers, we avoid confusion and achieve additional insight into the conditions that cause state to appear. My hope is that this additional insight will lead to an improvement in the terminology used to describe state to embedded observers, and so lead to improved programming language constructs for building systems in which state appears.

1.2 Modeling State

In order to study computational systems in which the phenomenon of state appears, we will need a way to model them. I will present the Connection Graph model and show how it can serve this purpose. Connection graphs were originally developed as an abstract machine model for a programming language for parallel computers.[4, 3]

Connection graphs have a number of advantages as a model of computation, particularly when it comes to studying state:

Connection graphs are *simple.* Simplicity is an advantage since it reduces the complexity of any analysis made using the model. We will be able to easily prove some theorems that would be very difficult otherwise.

Connection graphs are *sufficient.* A good model must be able the exhibit the phenomenon we want to study. It is actually somewhat surprising that connection graphs can be used to model the phenomenon of state. At first glance, connection graphs have a distinctly functional appearance. However, I will show how systems can be constructed in which state occurs.

Connection graphs are *natural.* Real systems must correspond closely to their representation using connection graphs. In particular, since we are studying state as a perceptual phenomenon experienced by the *components* of a system, each component of a system must have a corresponding component in its connection graph model. (This rules out many universal computing models, such as Turing Machines and the Lambda Calculus. Such models are capable of simulating any computational system, but they don't preserve the system's structure as connection graphs do.)

1.3 Outline

In section 2 I shall begin by presenting the connection graph model and demonstrating how programs can be translated into connection graphs. This is done in some detail so that the reader may acquire a solid understanding of how ordinary programming language concepts are expressed in the simple connection graph language. I will pay particular attention to the problem of generating state-like behavior in the connection graph model.

In section 3 I will identify two phenomena that occur whenever connection graphs are used to model systems with state. I will argue that, given our intuitive understanding of state, these phenomena are expected symptoms of state-like behavior. Then, in section 4, I will prove some theorems about the conditions under which those phenomena can occur, and argue that those conditions are the correct characterization for systems that exhibit state-like behavior. Such systems are those that *depend* on their nonlocal topological structure in order to behave correctly.

In section 5 I will argue that much of our trouble with state stems from the mistaken notion that state can always be localized within some *object.* This object metaphor is the only tool supported by current programming languages for using state, and while it works in simple situations, it becomes very difficult to apply as systems become more complex. I will suggest that we should be searching for better metaphors, not involving the notion of objects, for controlling the nonlocal properties of our computational systems.

Finally, in section 6, I will propose that the next step towards exploiting these ideas should be the construction of a compiler that uses its understanding of state to achieve some of the benefits normally associated with purely functional programming systems.

2 The Connection Graph Model

In this section I present the basics of the connection graph model. I will show how to translate a program into a connection graph grammar, with particular attention to how state-like behavior can be implemented. A good feel for this translation, especially with respect to state, is needed in order to understand the discussion that follows.

2.1 Connection Graphs

Intuitively, a connection graph is similar to the topological structure of an electronic circuit. An electronic circuit consists of a collection of gadgets joined together by wires. Gadgets come in various types—transistors, capacitors, resistors, etc. Each type of gadget always has the same number and kinds of terminals. A transistor, for example, always has three terminals called the collector, the base, and the emitter. Each terminal of each gadget can be joined, using wires, to some number of other terminals of other gadgets.

A connection graph differs from a circuit chiefly in that we restrict the way terminals can be connected. In a connection graph each terminal must be connected to *exactly one* other terminal. (In particular, there can be no unconnected terminals.)

Symmetrical gadgets are also ruled out. Some gadgets found in circuits, such as resistors, have two indistinguishable terminals. In a connection graph all the terminals of any particular type of gadget must be different.

Some convenient terminology: The gadgets in a connection graph are called *vertices.* The type of a vertex is called simply a *type*, and the terminals of a vertex are called *terminals.* The wires that join pairs of terminals are called *connections.* The number of terminals a vertex has is its *valence.*

The type of a terminal is called a *label.* Thus, associated with each vertex type is a set of terminal labels that determine how many terminals a vertex of that type will possess, and what they are called. For example, if we were trying to use a connection graph to represent a circuit, the type TRANSISTOR might be associated with the three labels COLLECTOR, BASE, and EMITTER.

Given a set of types, each with an associated set of labels, we can consider the set of connection graphs *over* that set of types, just as given a set of letters called an alphabet, we can consider the set of strings over that alphabet. Figure 1 shows a connection graph over the two types TRANSISTOR and CONS, where type CONS has the three associated labels CAR, CDR, and UP. This example is a single connection graph—a connection graph may consist of several disconnected components.

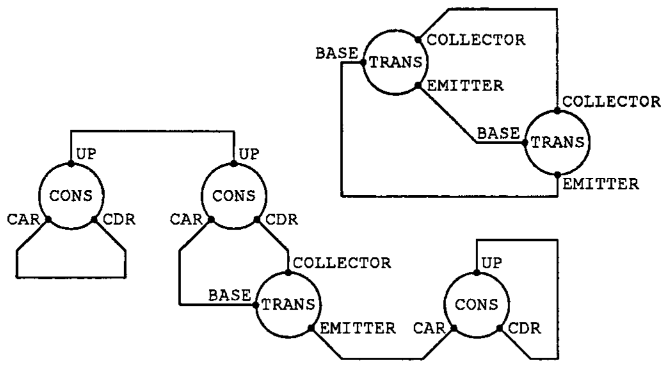

Figure 1: Example Connection Graph

Figure 2 is *not* an example of a connection graph; the UP terminal of the CONS vertex is not connected to exactly one other terminal. The restriction that terminals be joined in pairs is crucial to the definition.

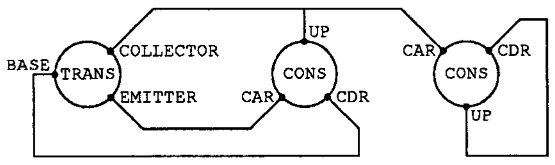

Figure 2: Illegal Connection Graph

2.2 Connection Graph Grammars

A *connection graph grammar* is a collection of production rules called *methods*. Each method describes how to replace a certain kind of subgraph with a different subgraph. If the connection graphs over some set of types are analogous to the strings over some alphabet, then a connection graph grammar is analogous to the familiar string grammar.

In a string grammar the individual rules are fairly simple, consisting of just an ordered pair of strings. When an instance of the first string, (the *left hand side*) is found, it may be replaced by an instance of the second string (the *right hand side*). It is clear what is meant by *replacing* one string with another.

In a connection graph grammar the notion of replacement must be treated more carefully. Figure 3 shows an example of a method. Both the left hand and right hand sides of a method are subgraphs with a certain number of *loose ends*. A method must specify how the terminals that used to be connected to terminals in the old subgraph should be reconnected to terminals in the new subgraph. In the figure, the loose ends in each subgraph are numbered to indicate how this reconnection is to be done.

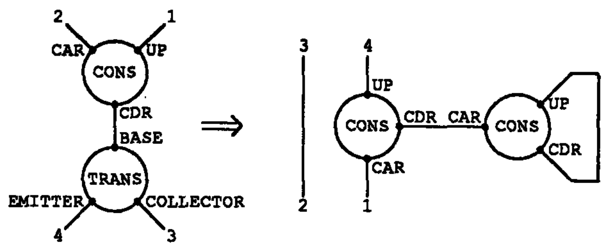

Figure 3: Example Method

For example, when applying the method in figure 3, a `CONS` vertex and a `TRANSISTOR` vertex, connected from `CDR` to `BASE`, are to be replaced with two new `CONS` vertices connected together as indicated. The terminal in the old connection graph that was connected to the `UP` terminal of the old `CONS` vertex is reconnected to the `CAR` terminal of the first new `CONS` vertex, as shown by the loose ends numbered 1. The terminal that was connected to the `EMITTER` terminal of the old `TRANSISTOR` vertex is reconnected to the `UP` terminal of the same new `CONS` vertex, as shown by the loose ends numbered

4. The terminal that was connected to the `CAR` terminal of the old `CONS` vertex, and the one that was connected to the `COLLECTOR` terminal of the old `TRANSISTOR` vertex, are reconnected to *each other*—this is indicated by the loose ends numbered 2 and 3.

It should be emphasized that the old subgraph—consisting of those vertices matching the left hand side of the method just applied—is discarded. In this aspect connection graph grammars really are exactly analogous to the way grammars are defined for strings.

It might be interesting to continue the analogy with string grammars by introducing a distinction between terminal and nonterminal types and identifying a initial nonterminal type. Then we could define the *language* generated by a given connection graph grammar as the set of connection graphs that can be generated by starting with a graph consisting of a single vertex of the initial type and applying methods until a graph with only terminal type vertices results. There might be interesting results to be proved, for example, about what kind of connection graphs can be generated using only *context sensitive* connection graph grammars, where that notion is suitably defined.

In using connection graph grammars as a model of computation we have no need of terminal and nonterminal types, nor of an initial type. We translate a program into a connection graph grammar, and then apply it to some *input* graph. After methods from the connection graph grammar have been applied until no more are applicable, some *output* graph will result. Thus the connection graph grammar may be viewed as computing some function from connection graphs to connection graphs.

Actually it is a multivalued function—more properly a relation—since the output connection graph can depend on the order in which methods from the connection graph grammar are chosen. Connection graph grammars are nondeterministic in general, but in section 4 we will be able to prove a Church-Rosser theorem for certain classes of grammars.

Only one form of method will appear in the connection graph grammars generated by the translation process described below: methods whose left hand side consists of exactly two vertices joined by a single connection. Figure 3 is an example of such a *binary method.*

2.3 Translating Lisp into a Grammar

A program written in a familiar, Lisp-like notation can be translated straightforwardly into a connection graph grammar. There are two key ideas involved: first, a natural graph-structure is already associated with simple expressions that don't involve LAMBDA-abstractions or conditionals, and second, a binary method can be used to implement a procedure calling mechanism.

The translation scheme given here assigns an "eager" applicative order semantics to the source language. It is applicative because the bodies of LAMBDA-expressions and the arms of conditional expressions are not processed in any way until the procedure is called or the test results are known. It is eager because procedures are called while their arguments continue to compute. (This is similar to the semantics that would result if Multilisp [8] were modified to implicitly wrap a FUTURE around all operands of all procedure calls.)

2.3.1 Expressions

It is plain how to interpret simple expressions as the description of a graph—we can just use the *expression tree.* Figure 4 shows the connection graph for the expression `(+ 3 (* 4 5))` using the vertex types CALL2, +, *, 3, 4, and 5. (CALL2 vertices are used to represent two-argument procedure calls. + and * vertices represent two well-known primitive procedures. 3, 4, and 5 vertices represent the integers three, four, and five. Similar vertex types will be introduced below without comment.)

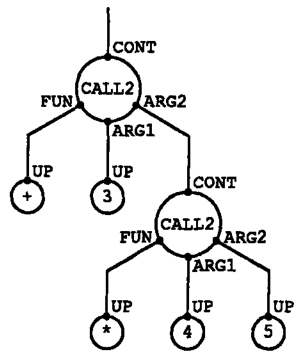

Figure 4: Graph of `(+ 3 (* 4 5))`

This isn't really a proper connection graph because it has a loose end at the top. Loose ends are an artifact of the way expressions can be nested. If the example expression appeared as part of some more complicated expression, then that loose end would be used to join the example graph into some more complicated graph.[1]

2.3.2 Variables and LAMBDA

Since the semantics of variables is intimately related to that of LAMBDA-abstraction, it is convenient to explain the interpretation of LAMBDA-expressions before taking up the interpretation of variables.

The connection graph for the expression

```
(F 2
   (LAMBDA ()
     (+ 3 (* 4 5))))
```

is shown in figure 5. G0069 is a unique vertex type generated to represent closures of the LAMBDA-expression. Associated with this vertex type is the method shown in figure 6. The right hand side of this method is just the graph represented by the body of the LAMBDA-expression. The left hand side is the graph fragment that would occur were a vertex of type G0069 to be invoked as a procedure of no arguments.

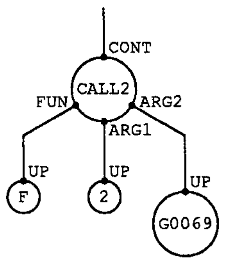

Figure 5: Graph of (F 2 (LAMBDA () (+ 3 (* 4 5))))

The introduction of anonymous vertex types, such as G0069, and new methods for those types, such as the method in figure 6, is a consequence of the declarative nature of LAMBDA-expressions. The graph of a LAMBDA-expression itself is always very simple, consisting of a single vertex of an

[1]An interesting discussion of the relationship between expressions and the networks or graphs they notate can be found in [14]. Note, however, that a connection graph is not the same thing as a constraint network; the notion of a two-ended connection differs from the more wire-like notion of identification used in a constraint language.

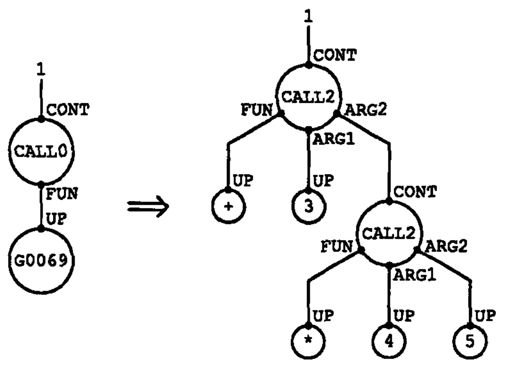

Figure 6: Method for G0069 Vertices

anonymous type. The body of the LAMBDA-expression declares how that type should behave in conjunction with an appropriate CALL vertex.

The handling of bound variables is an obvious extension. Consider now the expression

```
(F 2
   (LAMBDA (Y)
     (+ 3 (* Y 5))))
```

whose graph appears in figure 7. G0259 is again a vertex type generated to represent closures of the LAMBDA-expression. Associated with this vertex type is the method shown in figure 8. This method arranges that when a vertex of type G0259 is invoked as a procedure of one argument, that argument is connected to the place where the variable Y appeared in the graph of the body of the LAMBDA-expression.

Next, we would like to consider an expression like

```
(LAMBDA (Y)
  (+ X (* Y 5)))
```

that has free variables. Since a free variable is always bound by *some* surrounding contour, we consider instead the expression

```
(LAMBDA (X)
  (F 2
     (LAMBDA (Y)
       (+ X (* Y 5)))))
```

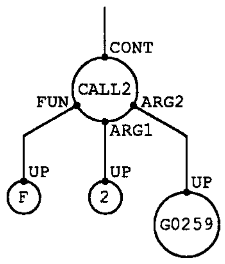

Figure 7: Graph of (F 2 (LAMBDA (Y) (+ 3 (* Y 5))))

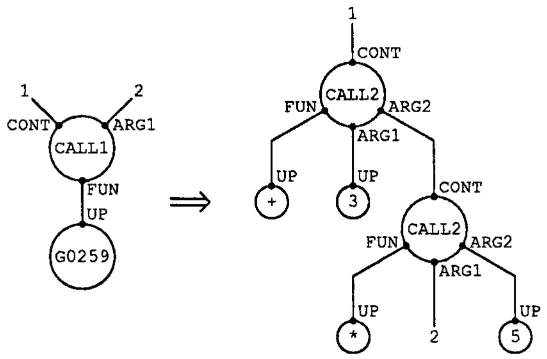

Figure 8: Method for G0259 Vertices

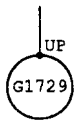

Figure 9: Graph of (LAMBDA (X) (F 2 (LAMBDA (Y) (+ X (* Y 5)))))

whose rather uninteresting graph appears in figure 9.

Associated with the vertex type G1729 is the method shown in figure 10, and associated with the vertex type G1776, which appears in the right hand side of that method, is the method shown in figure 11.

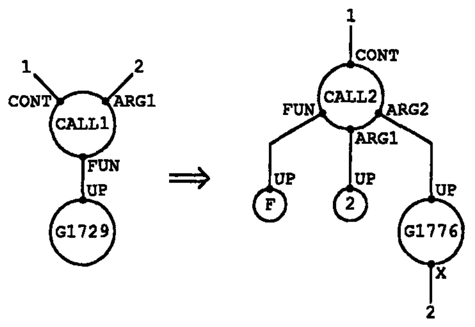

Figure 10: Method for G1729 Vertices

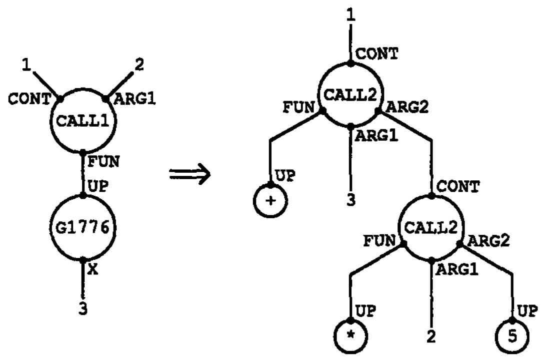

Figure 11: Method for G1776 Vertices

The vertex type G1776, generated to represent closures of the inner LAMBDA-expression, is bivalent—previously all such generated vertices have been univalent. The extra terminal is used to pass the value of the free variable X from where the closure was generated to where it is invoked.

Now we can see why we need only consider binary methods: they can be used to capture the basic mechanism of procedure calling. One vertex represents the procedure to be called. Its terminals (except the one connecting it to the other vertex) are the environment of the procedure. Its type is the procedure code. The other vertex is the argument list. Its terminals are the arguments to be passed to the procedure, and the continuation to be connected to the returned value. Its type is used to indicate the operation that should be performed (the procedure should be *called*), and allows a procedure call to be distinguished from a procedure that is merely connected to some static data structure, such as when a procedure is an element of a a list built from `CONS` vertices.

This suggests how a binary method can also be viewed as a *message pass* [7, 12]. One vertex is the object, the other is the message. The terminals of the object vertex are its instance variables. The terminals of the message vertex are the arguments to the message. The type of the object vertex is the type of the object. The type of the message vertex is the operation. Figure 12 shows the method for an object of type `4` receiving an `ADD1` message. The method dispatch normally associated with message passing occurs when we look up which method to run for the pair of vertex types in question. (This, of course, explains why I called them "methods" in the first place.)

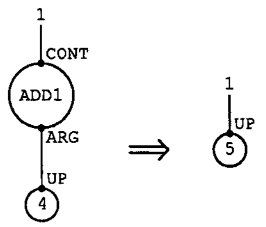

Figure 12: Method for Incrementing a 4

When a binary method is viewed as a message pass, the difference between the message and the object is entirely in the eye of the beholder. The method could just as well be interpreted in the other way, so that object and message exchange roles. This symmetry is possible because connections themselves are symmetrical. In ordinary programming languages, where all objects are referenced through asymmetrical pointers, this symmetry doesn't exist.

Figure 12 raises a minor issue about the relationship between message passing and procedure calling that seems to confuse many people. The expression `(ADD1 4)` describes a procedure call rather than the message pass that appears on the left hand side of figure 12. If the ADD1 procedure is defined appropriately, however, the graph of (ADD1 4) will trivially transform into the left hand side of figure 12.

2.3.3 Conditionals

The translation of the conditional expression

```
(IF (EVEN? 3)
    4
    5)
```

is shown in figure 13. G1957 is a unique vertex type generated to test the value returned by the expression (EVEN? 3). Figure 14 shows the two methods associated with type G1957. The first method covers the case where three is even by returning four. The second method returns five in the case where three is odd.

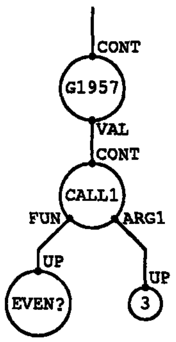

Figure 13: Graph of (IF (EVEN? 3) 4 5)

Another example demonstrates the interaction of conditionals with variables. Figure 15 shows the method associated with the vertex type G1984, generated to represent closures of the LAMBDA-expression

```
(LAMBDA (X Y)
  (IF (EVEN? X)
      Y
      (* 2 Y))) .
```

The vertex type G2020 is used to test the value returned by the expression (EVEN? X). Since the variable Y appears in both arms of the conditional,

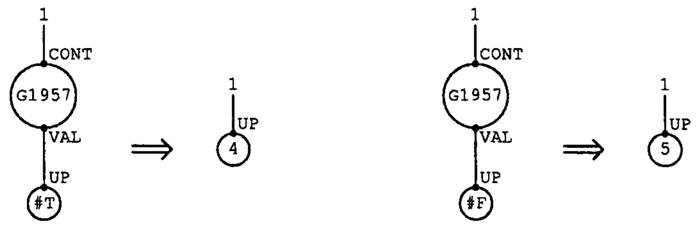

Figure 14: Methods for G1957

a G2020 vertex must have an additional terminal to attach to the second argument to the procedure so that the methods for G2020 vertices (shown in figure 16) can use it.

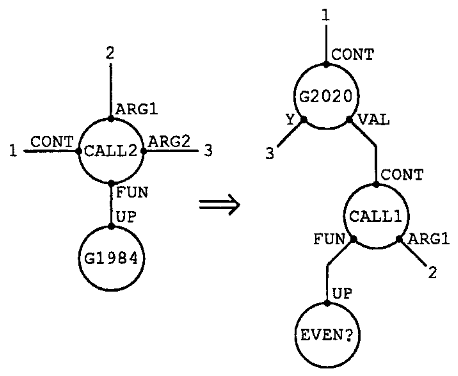

Figure 15: Method for (LAMBDA (X Y) (IF (EVEN? X) Y (* 2 Y)))

Notice that this is *not* equivalent to treating IF as a macro that would expand into

```
(IF-PROCEDURE
  (EVEN? X)
  (LAMBDA () Y)
  (LAMBDA () (* 2 Y)))
```

(where IF-PROCEDURE is defined to invoke its second or third argument depending on the value of its first). This would create two vertices of generated type to represent closures of the two LAMBDA-expressions. Since the variable

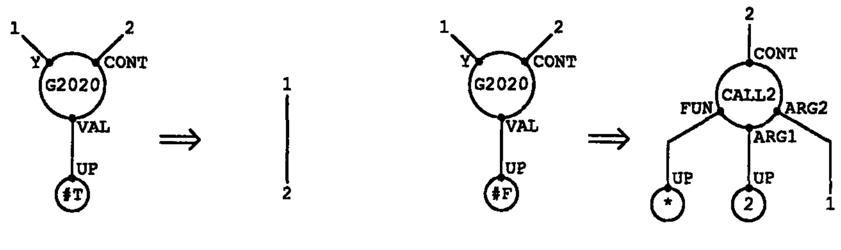

Figure 16: Methods for G2020

Y appears free in both expressions, both vertices would be bivalent: their extra terminal would be used to pass in the value of Y. Thus we would have to somehow make a *copy* of the value of Y so that it could be connected to both vertices.

The problem of copying is dealt with in the next section, but we will have reason to see that it is not a desirable phenomenon. Using an IF-PROCEDURE to translate conditionals breaks the conditional into two parts: a dispatching part, and an invocation part. By keeping these aspects of conditional expressions bound together, the copying problem can be avoided in this case.

2.3.4 Copying

Consider the expression

```
(LAMBDA (X)
  (+ X X)) .
```

Because X occurs twice in the body, the method associated with the vertex type generated to represent this procedure must arrange to distribute a single argument to two places.

We could simply outlaw such procedures, obtaining a language in which the programmer is forced to specify exactly how each reference will be duplicated. This would be a great inconvenience, and would violate many people's intuitions about the meaning of expressions, especially in such straightforward operations as arithmetic, where it is perfectly clear what it means to duplicate a integer.

Figures 17 and 18 suggest a solution. The method in figure 17 uses a COPY vertex to increase the fan-out of the procedure's argument. In the general case, a tree of trivalent COPY vertices will be constructed. The method in figure 18 implements the usual semantics for duplicating an integer. An infinite collection of such methods, one for each integer, are needed.[2] Methods for the duplication of other objects can be constructed by the programmer on a type-by-type basis.

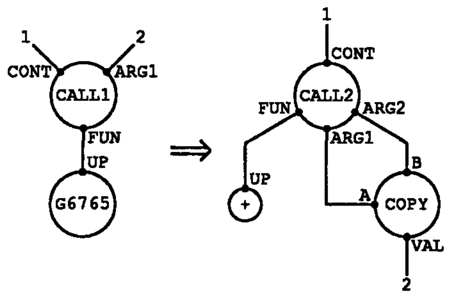

Figure 17: Method for (LAMBDA (X) (+ X X))

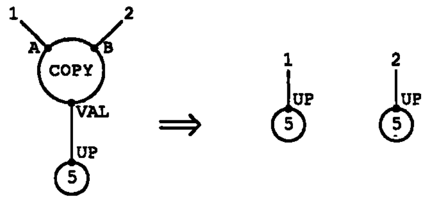

Figure 18: Method for Duplicating 5

2.4 State-like Behavior

Given the presentation of connection graphs up to this point, it is perhaps surprising that they can be used to construct systems that behave as if they had state. The key idea is that instead of using COPY vertices to duplicate structure, as was done in the last section, we use COPY vertices to construct communications networks. A object that appears to have state can be implemented by allowing the COPY vertices to accumulate into a fan-in tree with the state stored at the apex (figure 19).

[2]In a practical implementation of the connection graph model, these methods would all share the same code. But such tricks needn't concern us here.

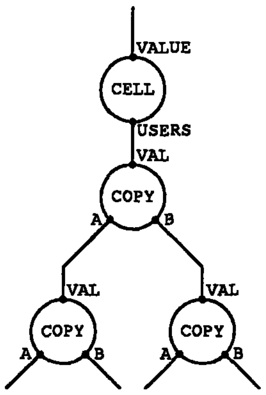

Figure 19: Fan-in Tree

Instead of writing methods that *duplicate* the structure at the apex, methods can be written that propagate messages up the tree where they can then *interact* with that structure. Messages arrive serially at the apex of the tree. As each message arrives, the structure can transform into a different structure before handling the next message. The structure connected to the apex of such a tree can be thought of as its current state.

Figures 20, 21 and 22 show six methods that use this technique to implement an object called a CELL that responds to PUT and GET operations to modify the contents of a state variable.

The four methods in figure 20 are variations of the same basic idea. They allow PUT and GET operations to propagate from multiple references at the leaves, up the fan-in tree, to the apex, where the state variable can be accessed. Each operation requires two separate propagation methods: one for when the operation is connected to the A terminal of a COPY vertex, and another for the B terminal. PUT and GET vertices have a USERS terminal to hold the part of the tree they have propagated past.

The method in figure 21 causes the GET operation to return a copy of the value stored in the CELL vertex when it encounters it at the apex of the fan-in tree.

Figure 22 causes the PUT operation to replace the value stored in the cell and drop the old value. DROP vertices are used to dispose of unwanted connections just as COPY vertices are used to multiply access to popular ones.

Figures 23 through 26 demonstrate how a CELL functions. Initially (in the

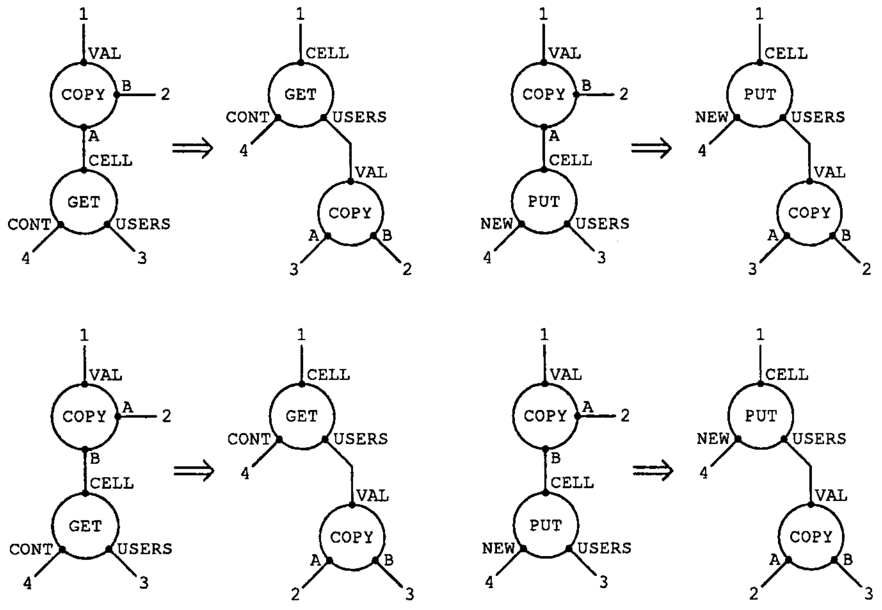

Figure 20: Methods for Propagation

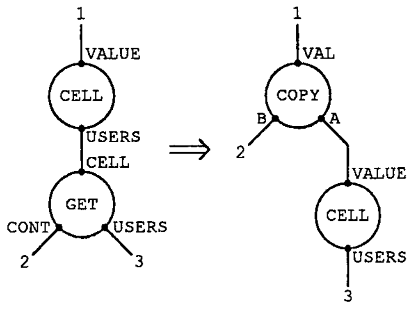

Figure 21: GET Method for a CELL

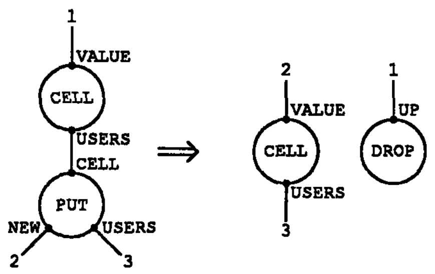

Figure 22: PUT Method for a CELL

left half of figure 23) there are two messages, one PUT and one GET, waiting at the fringe of a small (single vertex) fan-in tree of COPY vertices with a CELL at its apex. This connection graph might result from the expression

```
(LET ((X (CELL 3)))
  (IN-PARALLEL (PUT X 4)
               (GET X)))
```

assuming that the procedures PUT, GET, and CELL are given the appropriate trivial definitions to construct PUT, GET, and CELL vertices.

There are two methods from figure 20 that can be applied to the left connection graph in figure 23. I arbitrarily chose to apply the method for propagating the PUT message through the fan-in tree first. Had I chosen to propagate the GET, a different final graph would have been the result. Presumably the programmer who wrote this program didn't care which of the two possible answers, 3 or 4, he obtained.

In figure 24, where there are also two applicable methods, the method for delivering a PUT message to a CELL (figure 22) is chosen instead of propagating the GET. In this case the choice does not change the final answer.

In figure 25 only the method for propagating the GET can be applied, and then in figure 26 the sole possibility is to deliver the GET message to the CELL. (The resulting debris of COPY and DROP vertices can be cleaned up by some methods for combining DROP and COPY vertices, dropping and duplicating integers, and dropping CELL vertices. These methods are easy to construct, but are not given here.)

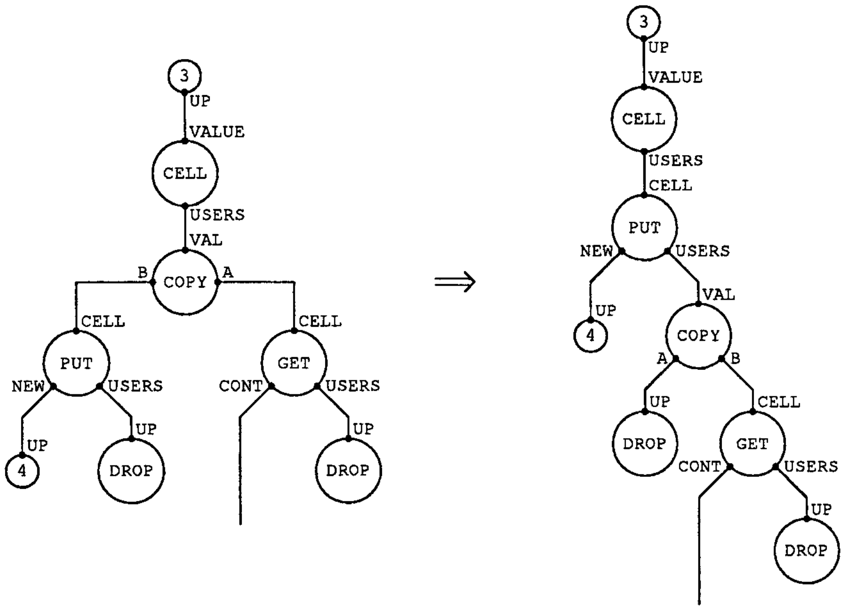

Figure 23: Propagate PUT

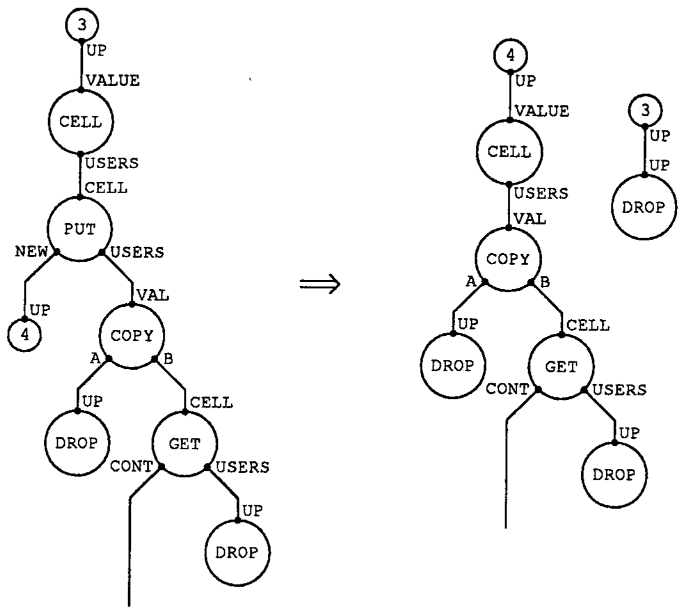

Figure 24: Perform **PUT**

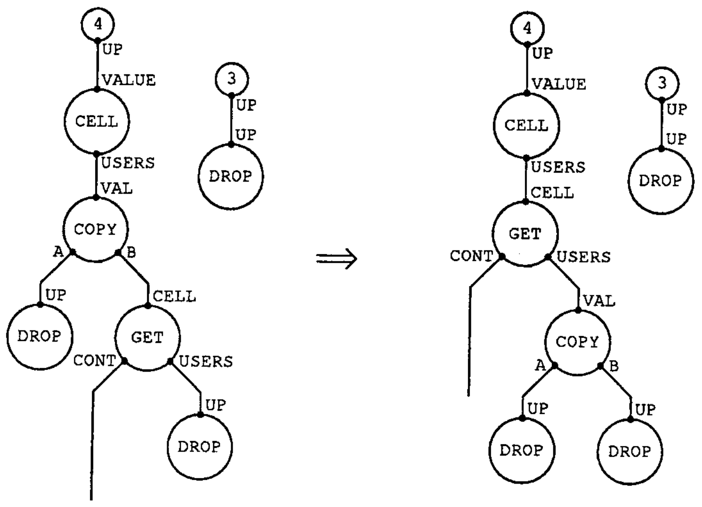

Figure 25: Propagate **GET**

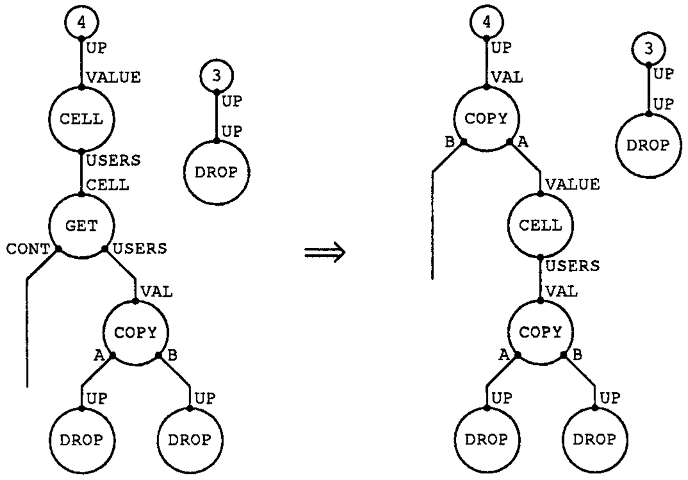

Figure 26: Perform GET

3 The Symptoms of State

Experience with converting programs into connection graph grammars has convinced me that all grammars that exhibit the phenomenon of state share two important characteristics: First, they are always nondeterministic grammars. Second, they always construct graphs that contain cycles. In this section I shall present some intuitive arguments for why this should be so. In the next section I will show why these two characteristics constitute strong circumstantial evidence that state is a phenomenon caused by the nonlocal topological structure of connection graphs.

It would be nice to be able to *prove* that the phenomenon of state has this topological origin. Unfortunately this cannot be done because state is not something that has been adequately defined. All programmers understand what state is because they have experienced it in the systems they construct. They know it when they see it, but they don't have a formal definition for it. Thus, the best that we can hope to do is to demonstrate that this topological property exhibits the same *symptoms* that we normally associate with state. We cannot show that some new definition of state is equivalent to some known definition, but we can give state a definition for the first time.

3.1 Symptom: Nondeterminism

Why should it be the case that nondeterministic connection graph grammars are needed in order to construct systems with state?

Recall that in figure 23 an arbitrary choice was made to propagate the PUT message up the fan-in tree of COPY vertices towards the CELL at the top. Had the GET message been propagated instead, a different final connection graph would have resulted. This nondeterminism is possible because COPY vertices are willing to interact with vertices connected to either their A or B terminals. Intuitively, a fan-in tree is willing to "listen" to messages arriving from anywhere along its fringe.

If COPY vertices were only willing to interact through their A terminals, there would only be a single location on the fringe of a fan-in tree that would ever be listening for the next message. This single attentive location would be determined at the time the tree was constructed. For each variable that occurred twice in the body of a LAMBDA-expression, programmers would have to declare which occurrence was the A occurrence. In effect, programmers would have to specify the order in which each reference to an object would be used. This is the price we would have to pay for keeping our grammars deterministic.

To illustrate the effect this would have, consider the following simple Scheme[13] procedure for computing factorials:

```
(DEFINE (FACT1 N)
  (DEFINE (LOOP N X)
    (COND ((< N 2) (GET X))
          (ELSE
           (PUT X (* (GET X) N))
           (LOOP (- N 1) X))))
  (LOOP N (CELL 1)))
```

This program uses a simple CELL object to accumulate a product as it loops over the integers from N down to 2. (Assuming that CELL's are constructed by applying the CELL procedure to an initial value, and that that value can then be read and written using the procedures GET and PUT.)

No Lisp programmer would ever actually write FACT1 since it is easy to see that it is equivalent to the following, clearly state-free, procedure:

```
(DEFINE (FACT2 N)
  (DEFINE (LOOP N A)
    (COND ((< N 2) A)
          (ELSE
           (LOOP (- N 1) (* A N)))))
  (LOOP N 1))
```

To prove that FACT2 behaves the same as FACT1 one must reason about the history of the operations performed by FACT1 on its CELL: Only a single CELL is created, and it initially contains 1. Each time around the loop the CELL's contents are first read, and then written. At the end of the loop, the CELL's contents are read one final time. Given this history it is simple to replace the CELL in FACT1 with the variable A in FACT2. As FACT2 loops, the value of A mimics the contents of the CELL in FACT1.

Now imagine translating FACT1 into a connection graph grammar. The variable X occurs three times in the ELSE clause of the COND statement. (This is the only multiple occurrence of any variable in FACT1.) It would seem therefore that any translation of FACT1 into a connection graph grammar would require the use of nondeterministic COPY vertices.

However, the semantics of the Scheme language require that when the ELSE clause is evaluated, the next operation performed on the CELL, the GET operation, will use the reference obtained from the second occurrence of X. The PUT operation will then use the reference obtained from the first occurrence of X, and all following operations will use copies of the reference obtained from the third occurrence of X. Thus we can order the occurrences of X in FACT1 so as to specify the order in which references obtained from them will be used. This enables us to translate FACT1 into a *deterministic* connection graph grammar using COPY vertices with a single attentive terminal.

The knowledge needed to prove that FACT2 is equivalent to a stateless procedure turns out to be similar to the knowledge needed to translate FACT2 into a deterministic grammar. The former required knowledge of the history of operations performed on the CELL. The latter required knowledge of the order in which the occurrences of X would be used. Since each occurrence of X is clearly tied to a particular operation (the first is an argument to PUT and the second is an argument to GET) these are really just two manifestations of the same knowledge. Augmenting FACT1 with the declarations necessary to allow it to be translated into a deterministic connection graph grammar would be augmenting it with the first half of the argument that it can be rendered stateless.

If FACT1 truly needed to use state—that is, if it wasn't equivalent to some procedure like FACT2—it would be very surprising to find that we could add the ordering declarations that would allow it to be translated into a deterministic grammar. Given such declarations, it takes only a little work to find an equivalent stateless procedure. So if a procedure has no stateless equivalent, it shouldn't be possible to add such declarations. And without such occurrence ordering declarations, we can only produce a nondeterministic grammar.

The foregoing argument falls short of being a rigorous proof that state implies nondeterminism. We do not have a definition for *state*, nor do we have a rigorous notion of *equivalence* for procedures. Nevertheless it is clear that there is a close relationship between what needs to be done to a connection graph grammar to make it deterministic, and what needs to be done to a procedure to make it stateless.

Others have also observed that the need for nondeterminism seems to be a symptom of the desire for state. In [9], for example, Henderson must add a nondeterministic stream merging operator before he can construct an otherwise functional description of an operating system that appears to maintain state.

3.2 Symptom: Cycles

Why should it be the case that cyclic connection graphs are needed in order to construct systems with state?

Consider how an observer embedded in such a system can perceive state. There must be some experiment that the embedded observer can perform that will reveal that the part of the connection graph external to him behaves as if it had state.

Such an experiment, expressed in Scheme, might look like:

```
(DEFINE (EXPERIMENT X)
  (ACTION! X)
  (IF (TEST? X)
      'STATE
      'NO-STATE)) .
```

The general idea is to detect that the external system, accessed through the variable X, somehow *remembers* the action performed by ACTION!, and this can be detected by the procedure TEST?. The programmer who wrote this procedure probably thought in terms of some *object*, named by the variable X, whose state could be modified and tested by ACTION! and TEST?.

The important thing to notice about the procedure EXPERIMENT is that the variable X occurs in its body twice. This is because two references to the subsystem being tested are needed in order to complete the experiment. While one reference is passed to ACTION!, a second reference must be retained so that ACTION!'s effects can be observed.

If EXPERIMENT were translated into a connection graph grammar, then, due to the two occurrences of X, the first method would resemble figure 17 in that its right hand side would contain a cycle. This cycle is not a spurious effect of the way the procedure was written, it it a consequence of the nature of the experiment. Any system that looks for correlations between past actions and future effects will have this structure at some point in its history.

To see this more clearly, it may help to think about the phenomenon of *aliasing.* Aliasing occurs in traditional programming languages when a given storage location comes to have multiple names. Aliasing is often associated with puzzles that involve the way assignment interacts with different parameter passing mechanisms. When a location has multiple names, it becomes possible to change the value accessed through one name by using a another name. Thus, the behavior of an aliased name can be altered without ever using that name. It requires at least two names for this phenomenon to occur: a first name whose behavior changes mysteriously, even though it *wasn't* used, and a second name that causes the change because it *was* used.

If a name is viewed as a *path* for accessing a location, then the analogy with cyclic connection graphs is revealed. If there are two paths from point A, where the observer stands, to point B, the observed location, then there is a cycle starting from A, running down the first path to B, and then back up the second path to A again. Traversing the second path *in reverse* to get from B back to A may seem unnatural because we don't usually travel from objects backwards to the entities that know their names, but when modeling such a system using connection graphs it is easier to think in terms of cycles, a natural topological property of any kind of graph with undirected edges.

The need for cycles in systems with state has been noticed before. Usually it is expressed as a need for some kind of equality predicate in order to have a sensible notion of side effect. In [15] Steele and Sussman conclude that "the meanings of 'equality' and 'side effect' simultaneously constrain each other"; in particular they note that it is impossible to discuss side effects without introducing some notion of sameness.

The programmer who wrote the EXPERIMENT procedure intended that the variable X should refer to the *same* object each time it occurred; he was unable to discuss side effects using a notion of sameness. To support this notion we have to introduce cycles into the system. Cycles are thus inevitable when side effects are to be detected.

4 Locality

I will now demonstrate that the symptoms ascribed to state in the previous section occur in systems whose nonlocal topological structure affects their behavior. This suggests that the various phenomena we have loosely been calling "state-like behavior" can all be explained in these topological terms. This insight into the nature of state helps explain why programming in the presence of state is sometimes difficult, and why this difficulty increases as systems become larger. It also suggests where to look for further insights, and how we might design better tools for using state.

In this section the simplicity of the connection graph model will pay off in a big way. So far the restricted nature of connections has manifested itself chiefly by forcing us to construct the somewhat clumsy COPY vertices in certain situations. Here we will find that the simplicity of connections makes it very easy to define an appropriate notion of *locality*.

We need to capture the notion of locality because we can only study state as a phenomenon experienced by observers embedded in computational systems, and the only tool that an observer embedded in a connection graph has for making an observation is the binary method, whose left hand side is matched against a *local* subgraph. If there were methods whose left hand sides were more complex, perhaps allowing the method to run only if the entire graph passed some test, then locality would not be as important, but the left hand side of a binary method only tests a small, local portion of the graph. Thus, there is no way for a running program to learn anything about the nonlocal structure of the connection graph that it is a part of. With the characterization of locality developed below, this observation will be made precise.

It is worth recalling, at this point, that message passing and procedure calling are easily modeled using binary methods. Just as binary methods are unable to gain nonlocal knowledge, so message passing and procedure calling are similarly limited. This limitation is a consequence of the way the processing elements in all computing hardware work. All processing elements have some limit to the amount of state that can be contained in their private, immediately accessible memory. They are forced to take computational action based solely on this local knowledge of the state of the entire system. They must trust that other parts of the system—memories, other processing elements, I/O devices—are functioning as expected.

4.1 Homomorphisms and Local Indistinguishability

To capture the notion of locality, we can define a *homomorphism* from one connection graph to another as a map that preserves the local structure of the graph. More precisely, a homomorphism is a map $\vartheta : G \to H$ from the terminals of the connection graph G to the terminals of the connection graph H such that:

- If a and b are terminals in G, and a is connected to b, then $\vartheta(a)$ is connected to $\vartheta(b)$.
- If a and b are terminals in G that belong to the same vertex, then $\vartheta(a)$ and $\vartheta(b)$ belong to the same vertex in H.
- The label of a terminal a in G is the same as the label of $\vartheta(a)$ in H, and the type of a's vertex is the same as the type of $\vartheta(a)$'s vertex.

ϑ is an *epimorphism* if it is onto, a *monomorphism* if it is one-to-one, and an *isomorphism* if it is both. If ϑ is an isomorphism, it has an inverse ϑ^{-1} that is also an isomorphism.

Since all the terminals of a vertex in G are mapped together to the same vertex in H, a homomorphism also defines as a map from vertices to vertices. Thus if v is a vertex in G, we can extend our notation and let $\vartheta(v)$ be the corresponding vertex in H. In fact, a homomorphism is completely determined by its action on vertices.

Figure 27 shows an example of a homomorphism. The arrows indicate how the vertices of the left hand graph are mapped to the vertices of the right hand graph. This is the only homomorphism between these two connection graphs, although in general there may be many.[3]

Imagine what it would be like to explore a maze that was built on the plan of a connection graph: Each vertex becomes a room, each connection becomes a hallway, a sign over each doorway gives the label of the corresponding terminal, and sign in the center of each room gives the type of the corresponding vertex. Unless he turns around and retraces his steps, an explorer can never know that he has arrived in a room that he passed through

[3]The category of connection graphs and connection graph homomorphisms has many interesting properties. An entertaining exercise is to determine how to compute products of connection graphs.

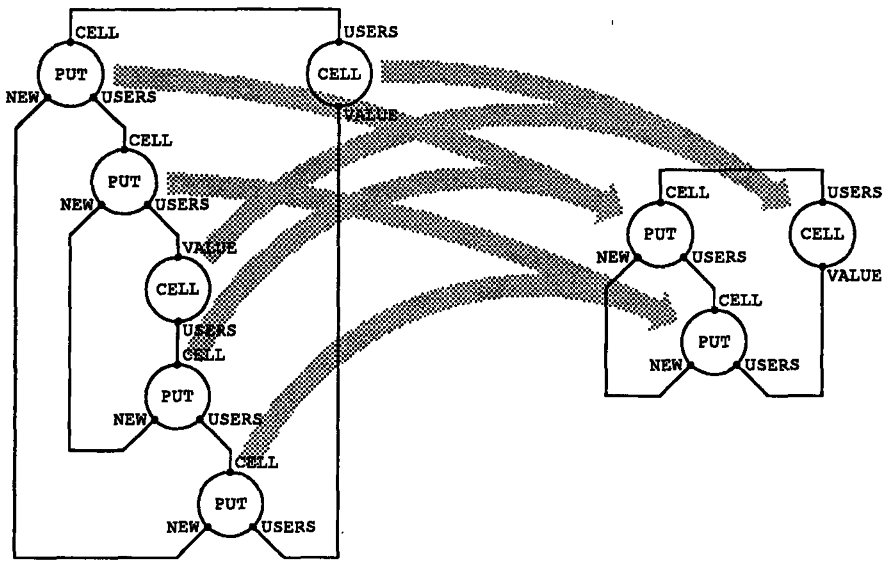

Figure 27: A Connection Graph Homomorphism

before. For all the explorer can tell, the connection graph he is exploring might well be a (possibly infinite) tree containing no cycles whatsoever. There would be no way for him to distinguish between the two connection graphs in figure 27. Such graphs are locally indistinguishable.

Formally, two connection graphs H_1 and H_2 are ***locally indistinguishable***, written $H_1 \sim H_2$, if there exists a connection graph G and epimorphisms $\vartheta_1 : G \twoheadrightarrow H_1$ and $\vartheta_2 : G \twoheadrightarrow H_2$. It can be shown that local indistinguishability is an equivalence relation on connection graphs. As a special case of this definition note that if $\vartheta : G \twoheadrightarrow H$ is any epimorphism, then $G \sim H$.

4.2 Methods

Things become more complicated once we introduce methods into the picture. In this section we will prove some theorems about the relationship between connection graph grammars and homomorphisms and local indistinguishability. The proofs are sketched rather than being presented in tedious detail, since the results are all easy to see once the definitions are understood.

We write $G \Rightarrow G'$ if the connection graph G' is the result of applying any number of methods to any number of *disjoint* subgraphs of G. We write $G_0 \Rightarrow^* G_n$ when there is a series $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$.

Theorem 1 *Given a homomorphism $\vartheta : G \rightarrow H$, and if $H \Rightarrow H'$, then there exists a connection graph G' and a homomorphism $\vartheta' : G' \rightarrow H'$ such that $G \Rightarrow G'$. This can be summarized in the following diagram:*

$$\begin{tikzcd}
G \arrow[r, "\vartheta"] \arrow[d, Rightarrow] & H \arrow[d, Rightarrow] \\
G' \arrow[r, "\vartheta'"] & H'
\end{tikzcd}$$

If ϑ is an epimorphism, then ϑ' can be found so that it is also an epimorphism.

Proof. Each occurrence of the left hand side of a method in H that is replaced in forming H' can be lifted back through ϑ to a set of occurrences of the same left hand side in G. The set of all such occurrences can then be replaced by the corresponding right hand sides to obtain G'. ϑ' can then be constructed from ϑ in the obvious way. □

Theorem 2 *Given $\vartheta : G \rightarrow H$, and if $H \Rightarrow^* H'$, then there exists G' and $\vartheta' : G' \rightarrow H'$ such that $G \Rightarrow^* G'$. If ϑ is an epimorphism, then ϑ' can be found so that it is also an epimorphism.*

Proof. This follows easily from the previous theorem by induction. □

The previous theorem is true given any connection graph grammar. It is the strongest such result I have found that does not constrain the grammar. For certain classes of grammars, and in particular for the class that contains most deterministic grammars, stronger theorems can be proven:

A connection graph grammar is *preclusive* if two occurrences of the left hand sides of methods can never overlap. This means that if the left hand side of a method ever appears in a connection graph G, and if $G \Rightarrow G'$, and if that subgraph was not replaced by the right hand side of the method in G', then that subgraph still appears in G'. The appearance of the left hand side of a method in a graph thus *precludes* the possibility that anything else will happen to those vertices before the method can be applied.

For example, any grammar that contains the methods in figure 20 cannot be preclusive. This is because it is possible to construct graphs such as the left hand graph in figure 23, where the COPY vertex belongs to two different subgraphs that match the left hand sides of different methods. We have already identified this property of COPY vertices in this grammar as a source of nondeterminism. The following theorem demonstrates that preclusive grammars are in fact deterministic.

Theorem 3 *If a connection graph grammar is preclusive, then given connection graphs G, G_1, and G_2 such that $G \Rightarrow G_1$ and $G \Rightarrow G_2$, there exists a connection graph G' such that $G_1 \Rightarrow G'$ and $G_2 \Rightarrow G'$.*

Proof. Since the grammar is preclusive the occurrences of method left hand sides in G are all disjoint. We can divide them up into four classes, (1) those that were replaced when forming both G_1 *and* G_2, (2) those that were replaced *only* in G_1, (3) those that were replaced *only* in G_2, and (4) those that were replaced in *neither*. Subgraphs in the second class must still occur in G_2, and subgraphs in the third class must still occur in G_1, so by applying the corresponding methods we can form G' from either G_1 or G_2. (In fact, $G \Rightarrow G'$ because we can apply the methods that correspond to the first three classes of left hand side occurrences in G.) □

Theorem 4 *If instead we have $G \Rightarrow^* G_1$ and $G \Rightarrow^* G_2$, then there exists G' such that $G_1 \Rightarrow^* G'$ and $G_2 \Rightarrow^* G'$.*

Proof. This follows easily from the previous theorem by induction. □

Theorem 4 shows most clearly what it is about preclusive grammars that makes them behave deterministically. It gives us a condition under which we have a Church-Rosser theorem for connection graphs. It shows that no matter what order we choose to apply methods, we always achieve the same result. If it is possible to apply methods until a connection graph is produced to which no further methods can be applied, then that graph is unique.

For preclusive grammars, in addition to theorem 1, we have the following more powerful result:

Theorem 5 *If a connection graph grammar is preclusive, then given connection graphs G, H, and H' such that $G \sim H$ and $H \Rightarrow H'$, there exists connection graphs G'' and H'' such that $G \Rightarrow G''$, $H' \Rightarrow H''$ and $G'' \sim H''$.*

Proof. The most straightforward way to construct G'' and H'' is to let them be the results of applying *all possible* methods whose left hand sides occur in G and H. This is possible because these are all disjoint subgraphs (since the grammar is preclusive). Further, it must be the case that H' is the result of performing some subset of these replacements, so by performing the remainder we see that $H' \Rightarrow H''$. It is clear from the construction that $G'' \sim H''$. □

Theorem 6 *If a connection graph grammar is preclusive, then given connection graphs G, H, G', and H' such that $G \sim H$, $G \Rightarrow^* G'$ and $H \Rightarrow^* H'$, there exists connection graphs G'' and H'' such that $G' \Rightarrow^* G''$, $H' \Rightarrow^* H''$ and $G'' \sim H''$. This can be summarized in the following diagram:*

$$\begin{tikzcd}
G \arrow[r, phantom, "\sim"] \arrow[d, Rightarrow, "*"] & H \arrow[d, Rightarrow, "*"] \\
G' \arrow[d, Rightarrow, "*"] & H' \arrow[d, Rightarrow, "*"] \\
G'' \arrow[r, phantom, "\sim"] & H''
\end{tikzcd}$$

Proof. This follows from the previous theorem by induction and by using theorem 4. □

Theorem 6 is very similar in form to theorem 4; it would be identical if we replaced the "$\sim$" with "$=$". Theorem 6 shows that not only doesn't it matter what order we choose to apply methods, it doesn't even matter which locally indistinguishable connection graphs we choose to apply them to. A preclusive connection graph grammar is completely insensitive to the nonlocal structure of the system.

4.3 Implications for Programs

The theorems we have just seen have implications for what an embedded observer can learn about the system it is embedded in.

Suppose we are given a pair of connection graphs G and H, and we are asked to produce a connection graph grammar that can somehow distinguish between the two. First, we need to be precise about what we mean by "distinguish". We want to be able to run the grammar on G or H and then apply some test to determine if the system has learned how it was initialized. The test must be *local*, otherwise we could supply the empty grammar and specify that the test is simply graph equality. Thus we will include two distinguished vertex types, `WAS-G` and `WAS-H`, in our grammar, and specify that if a vertex of type `WAS-G` ever appears in the graph, then it will be understood that the grammar has decided that the initial graph was G, and similarly `WAS-H` will signal that the grammar has decided that the initial graph was H.

Now consider the case where there is an epimorphism $\vartheta : G \rightarrow H$. Suppose that given some grammar we have $H \Rightarrow^* H'$ and that H' contains a vertex of type `WAS-H`, then by theorem 2 there is a graph G' where $G \Rightarrow^* G'$ and an epimorphism $\vartheta' : G' \rightarrow H'$. Since ϑ' is an epimorphism, G' must also contain a vertex of type `WAS-H`. The grammar is thus capable of deciding that the initial graph was H even though it was applied to G. The grammar will therefore be in error (unless it happens that ϑ is an isomorphism). Thus no grammar can ever correctly learn that it was initially applied to H, although it is possible that it might learn that it was applied to G.

Putting this observation in somewhat more computational terms: If, in the course of some computation, a system finds itself in configuration H, and there is an epimorphism $\vartheta : G \rightarrow H$, then from that point onward there is nothing that the system can do that will allow it to discover that it had in fact been in configuration H and not in configuration G. It might discover that it had been in configuration G, and from this it could conclude that it had *not* been in configuration H, but it can never discover that it had been in configuration H.

If such a system is halted in configuration H, and reconfigured to be in configuration G, the system can perhaps "malfunction" by arriving at a configuration G' (where $G \Rightarrow^* G'$) where there are no configurations G'' and H'' such that $G' \Rightarrow^* G''$, $H \Rightarrow^* H''$ and $G'' \sim H''$. (Such a malfunction might be called a "false step".)

There are two conditions under which such malfunctions are impossible:

- If the grammar is preclusive, then since $G \sim H$ theorem 6 guarantees us that if $G \Rightarrow^* G'$ we can find the requisite G'' and H''.

- If H contains no cycles, then any epimorphism $\vartheta : G \rightarrow H$ must be an isomorphism, so we can let $G'' = H'' = G'$.

Thus, replacing H with the locally indistinguishable G can cause a malfunction *only* if H contains cycles *and* the grammar is not preclusive. These are the two symptoms we previously identified as always being present in systems that exhibit state-like behavior.

This leads me to propose that systems that exhibit state-like behavior are, in fact, precisely those systems which depend on their nonlocal structure in order to function correctly.

This says a great deal about why programming in the presence of state should be difficult. Programming with state means that there are conditions which the system depends upon, that it is impossible for it to check.

To make this more concrete, recall the parable of the robot interacting with the computer via a keyboard and display. Remember that this was a system in which all components perceived state even though they could all be expressed in functional terms. Suppose we halt this system and replace it with a locally indistinguishable system. Specifically, replace it with a system consisting of two robots and two computers, where the first robot types on one computer's keyboard, but watches the display of the *other* computer, while the second robot types on the other keyboard and watches the first display.

In order to remain locally indistinguishable from the original system, each robot and each computer must be placed in the same state as if the system was still singular. Each robot "believes" that he is alone, and that he is interacting with a single computer. Initially both robots continue typing away secure in this belief. They are unable to detect that they now operate in a doubled system because they both type exactly the same thing at the same time, and the computers respond identically with the appropriate output.

Suddenly a fly lands on one of the displays. The robot watching that display pauses briefly to shoo it away. The other robot then notices that his display doesn't reflect his last few keystrokes, while the first robot notices that his display reflects keystrokes that he was only *planning* on making right before the fly disturbed his concentration. Upon further experimentation the robots eventually discover their true situation.

The original singular robot had no way of *testing* that he was part of the singular system, nevertheless he depended on this fact in order to act sensibly. He trusted that there really was a single computer that was responding to his keystrokes, and that what he saw on the display represented its reactions. He trusted that the file he typed in today, really would reappear when he called it up on the display tomorrow.

If you asked him to explain just how the rest of the system was able to behave in that way, he would explain that "the computer has state". That is his explanation of how the situation appears to him as an embedded observer, but it isn't a very good explanation from our point of view. It even has built into it the presupposition that there is only a *single* computer.

We can see that the property of the system that really matters, the property that the robot accepts and depends on to function in the system without error, is the untestable assertion that the system's nonlocal structure is what the robot believes it to be, and not some locally indistinguishable equivalent.

5 Supporting State without Objects

In the previous sections we saw that systems in which state appears are systems whose nonlocal topological structure is important to their correct functioning. In order to write correct programs that describe such systems, programmers must understand, and reason about, nonlocal properties. Unfortunately programming languages do not give programmers very much help in this task.

Most programming languages support only the metaphor of *objects* for using state. The simplest languages give the programmer *state variables*, simple objects that can be read and written. More advanced languages provide abstraction mechanisms that support the construction of abstract objects [7, 11, 12, 1] which support more complex operations.

The object metaphor is that objects serve as *containers* for state. Each container divides the system into two parts, consisting of the users of the container, and the keepers of the container. If the container is a state variable, the keepers will consist of memory hardware. If the container is some more complex object, the keepers will be implemented in software just like the users.

The users communicate with the keepers by passing notes through the bottleneck that is the object itself. The keepers are charged with maintaining the object metaphor. It is the keepers, for example, who must worry about making operations appear to happen atomically, should that be required. The keepers are slaves to the requirements of the users. They labor to maintain the illusion that the state is actually *contained in* the object.

The object metaphor works acceptably in many simple situations. It captures a commonly occurring pattern in which a component of a system serves as a clearinghouse of some kind. In order for such a clearinghouse to function, all of its users must know that they are using the *same* clearinghouse. This is an example of the kind of nonlocal structure discussed in the previous section. No experiment performed by those embedded in the system can determine that the system is configured correctly, but careful application of the protocols of the object metaphor confine the system to the correct nonlocal structure.

In more complex situations the object metaphor is less useful. If, for example, the keepers of object A must operate on object B in order to perform some operation, and the keepers of B then try to use A, incorrect behavior, such as a deadlock, can arise. The kinds of nonlocal properties that are needed to rule out such incorrect behavior are not captured well using the object metaphor alone. Additional reasoning (usually very special case reasoning) is required.

A better understanding of the nonlocal properties of computational systems might enable us to discover additional metaphors for thinking about state. If our programming languages supported these new metaphors, we could use them when appropriate, rather than being forced to phrase everything in terms of objects. Given a programming language that is sufficiently expressive about nonlocal properties, we would no longer need fear programming with state.

A possible approach to finding such new metaphors is to use the connection graph model to analyze systems in which the object metaphor has clearly failed to manage the system's state cleanly. Systems prone to various kinds of deadlock are not hard to find. An analysis of the nonlocal structure of deadlock-prone systems could point to new structures that can be used to avoid the problem.

6 The Next Step

The next step to be taken in order to exploit the ideas presented above is the construction of a demonstration system that achieves some of the benefits normally associated with functional programming systems, while still allowing the use of state.

A simple compiler will be written that translates programs written in a conventional Lisp-like language into a connection graph grammar. The grammar will be analyzed to discover which parts of the program are free from state-like behavior. The program will then be reconstructed as a program in *nearly* pure Lisp; the goal of the reconstruction will be to maximize the components of the output program that are written in pure Lisp, and to enclose the state-like behavior within known limits.

The reconstruction will also done with an eye towards executing the resulting code in parallel. While it is not strictly necessary to adopt this additional goal, its success can serve as a practical (and trendy) test for the utility of the system.

Although the functional subset of the target language will be traditional pure Lisp, it is unlikely that the target language will contain traditional impure Lisp procedures such as RPLACA and SETQ. Instead, primitive operations derived from the underlying connection graph model will probably prove more appropriate.

The source language will be as close to standard (not necessarily pure) Lisp as possible. This objective does not rule out language extensions and features that encourage the programmer to write code that the compiler will find easy to analyze. Indeed, such exercises in programming language design may prove to be the most valuable part of the endeavor. Language design cannot usefully be carried out in a vacuum; the expressiveness of a programming language must be tested somehow. In this case, the measure of the language's utility will be how well it functions in explaining state-like behaviors to a compiler that has a solid understanding of the causes of such behavior.

The overall approach thus attacks the problem of programming with state from two directions: first as an optimization problem, where a compiler works to eliminate hazards caused by the use of state, and second as a language design problem, where the programming language must allow the programmer to effectively communicate with the compiler about state.

The proposed system can achieve several different levels of success. In the trivial case, when the programmer writes purely functional code, the compiler will have no trouble producing purely functional output. The previous sections have demonstrated how a static analysis of a connection graph grammar can determine that no state-like behavior is possible. That a grammar is preclusive can be checked merely by examining the left hand sides of all the methods to see if any overlap. Since the translation for LAMBDA-expressions given above always produces preclusive grammars, functional Lisp programs are easily recognized as such. It is unnecessary to think about cyclic structure at all in this case.

With a little more work, it should be possible to recover functional code even in cases where the source appears to make use of state. The reasoning employed previously in the analysis of the FACT1 procedure demonstrates how this can be done. This reasoning is really no different from the kind of side-effect analysis undertaken by conventional compilers. The same ordering constraints normally manipulated by such compilers are used here to construct a preclusive connection graph grammar, and then to reduce the program to functional code. Again, cyclic structure does not enter into the picture.

The real profits should come when the compiler employs the complete definition of state during optimization. For example, even given a non-preclusive connection graph grammar, an examination of the methods that create cyclic structure sometimes reveals that the non-preclusive rules will never apply to the part of the graph that contains the cycle, and thus the program can be reduced to pure Lisp. And even in cases where state cannot be entirely eliminated, similar reasoning might be employed to limit the scope of the possible state-like behavior. Whether such a compiler can really achieve something new in the way of optimization cannot, of course, be predicted in advance, but it would be surprising if a compiler cannot realize some additional benefits from a real understanding of the causes of state.

It is traditional, in ventures such as this, to implement the compiler itself in its own source language; this ensures that at least one large program can be supported by the resulting system. Since one of the goals is that the source language should be conventional Lisp as much as possible, it might be sensible to try that trick here as well. However, this may be difficult to do initially, since the source language will be a moving target while any significant programming language design is occurring. A more reasonable goal would be to aim for the eventual conversion of the compiler into the source language once it becomes relatively stable.

7 Conclusion

The goal is a better understanding of state so that we can find better programming language constructs for using it. To summarize the steps we have just taken towards that goal and our current position:

State is not something that appears as an independent attribute of an object. Observers embedded in systems *perceive* state in other components, but an external investigator cannot, in general, locate the state in specific components. The best that can be done is to characterize those systems in which state-like behavior occurs.

The connection graph model is a simple way to represent programs. When connection graphs are used to model systems in which state occurs, two characteristics appear to be necessary: the connection graph grammars are always nondeterministic, and the connection graphs themselves always contain cycles.

These are the conditions we should expect if the correct characterization for systems that exhibit state is that they are those systems that must rely on their nonlocal topology in order to function correctly. This explains why external observers are unable to assign location to state: state is intrinsically a nonlocal phenomenon. It also explains why writing programs that are to function in systems where state is present should be difficult: such programs must rely on aspects of the system that are untestable, and thus hard to express in a programming language.

The usual programming language metaphor for thinking about these nonlocal properties is the metaphor of objects. I have proposed that we search for metaphors that better reflect the fact that state is a nonlocal phenomenon, and not something that can be kept in a container. A good way to start looking for such metaphors would be to examine phenomena such as deadlock, that occur in systems when the object metaphor breaks down.

As the next step towards exploiting these ideas, I propose to construct a compiler that uses its understanding of state to achieve some of the benefits normally associated with purely functional programming systems. The source language will be a nearly conventional Lisp dialect, possibly extended so as to give the programmer additional ways to communicate with the compiler about state. The target language will be pure Lisp wherever possible, with additional primitives derived from the connection graph model used to support state-like behavior when necessary. The compiler's goal will be to allow more efficient parallel execution of the program by reducing the hazards posed by state-like behavior.

Acknowledgments

Discussions with Guy Blelloch, David Chapman, Tom Knight, John Lamping, Dave Moon, Jonathan Rees, Chuck Rich, Gerald Sussman, and Ramin Zabih have helped me formulate and clarify these ideas. All the remaining confusion is entirely my own fault. Erasmus and George Bear provided much necessary support.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[3] Alan Bawden. Connection graphs. In *Proc. Symposium on Lisp and Functional Programming*, pages 258–265, ACM, August 1986.

[4] Alan Bawden. *A Programming Language for Massively Parallel Computers.* Master's thesis, MIT, September 1984. Dept. of Electrical Engineering and Computer Science.

[5] B. DeWitt and N. Graham, editors. *The Many-Worlds Interpretation of Quantum Mechanics.* Princeton University Press, 1973.

[6] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proc. Symposium on Lisp and Functional Programming*, pages 28–38, ACM, August 1986.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[8] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proc. Symposium on Lisp and Functional Programming*, pages 9–17, ACM, August 1984.

[9] Peter Henderson. *Is It Reasonable to Implement a Complete Programming System in a Purely Functional Style?* Technical Report, The University of Newcastle upon Tyne Computing Laboratory, December 1980.

[10] Tom Knight. An architecture for mostly functional languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 105–112, ACM, August 1986.

[11] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[12] David A. Moon. Object-oriented programming with Flavors. In *Proc. First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, 1986.

[13] Jonathan Rees and William Clinger. *Revised^3 Report on the Algorithmic Language Scheme.* Memo 848a, MIT AI Lab, September 1986.

[14] Guy L. Steele Jr. and Gerald Jay Sussman. *Constraints.* Memo 502, MIT AI Lab, November 1978.

[15] Gerald Jay Sussman and Guy L. Steele Jr. *The Art of the Interpreter or, The Modularity Complex.* Memo 453, MIT AI Lab, May 1978.

[16] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.