Connection Graphs

Alan Bawden

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Introduction

When thinking about programming languages, it is important to choose an appropriate abstract machine model. Such an abstract model serves to modularize the programming language problem into two pieces: translation of some high level language into the language of the abstract machine, and implemention of the abstract machine on real hardware. This paper presents *connection graph grammars* as an abstract model for *parallel* computation.

In order to obtain a good modularization, an abstract machine model must satisfy three requirements:

- It must be an appropriate model of actual hardware. It should not make operations that are expensive to support on real hardware seem cheap.

  On a parallel computer without shared memory, where interprocessor communication is the principal expense, connection graph grammars can be executed cheaply. This is true in part because they can be executed using graph reduction techniques that are purely local in nature, but this locality is greatly enhanced by the use of low-overhead *connections* for building the graph structure. Connection graph grammars closely approximate the ways that processing elements must communicate in such a parallel machine.
- It must be simple. Programmers will need to understand the model so that they can write and debug programs. Compilers will need to easily manipulate the model to compile and optimize programs.

  Connection graph grammars will be seen to be extremely simple. Constructing a compiler for a language based on connection graphs is relatively easy. That the resulting language can prove acceptable to many programmers remains to be seen.
- Translation of familiar programming language notions into the model must be straightforward. Programmer's intuitions about the behavior of familiar constructs should not be unduly violated.

  Most of this paper is devoted to demonstrating how a programming language can be translated into a connection graph grammar. The use of connections in the model does have some curious consequences at the language level. Most importantly, variables cannot necessarily occur multiple times within a single expression without special arrangement. At first this seems like a rather stiff limitation, but on closer examination it proves more illuminating than limiting.

The principal trade-off is thus between the the appropriateness of connection graphs as a model for parallel machines, and the unorthodoxy of the connection graph programming language. We can make the implementation more natural, by using connections, if we are willing to experiment with some changes in our programming language.

As will be shown, these changes cast some seemingly unrelated issues, such as the nature of objects with state, the relationship between generic operations and the types of objects, and the interaction of side-effects and non-determinism with β-reduction, in a new light. It would at least be *interesting* to program in the connection graph universe.

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*. As in a circuit, 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**.

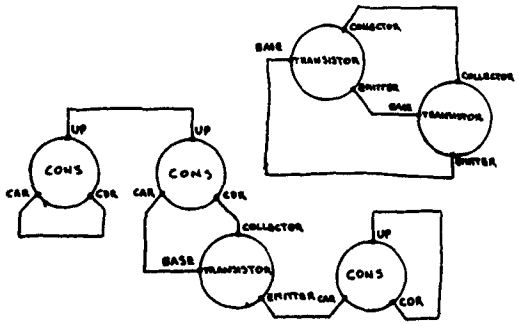

Figure 1. Example Connection Graph

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.

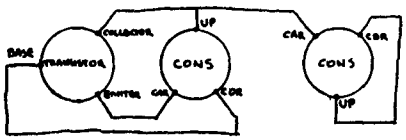

Figure 2. Illegal Connection Graph

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. It is this key property that makes connection graphs natural to implement on parallel hardware, and most profoundly influences the language.

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.

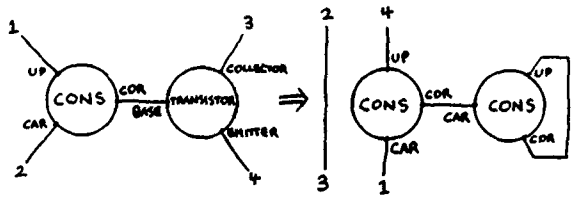

Figure 3. Example Method

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.

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 might be interesting to continue the analogy with string grammars by introducing a distinction between terminal and non-terminal types and identifying a initial non-terminal 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 parallel computation we have no need of terminal and non-terminal 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 are 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 multi-valued 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.)

Only one form of method will appear in the connection graph grammars generated: methods whose left hand side

consists of exactly two vertices joined by a single connection. Figure 3 is an example of such a *two-vertex method.*

Implementing Connection Graphs

I shall assume a fairly general description of a parallel computer: a collection of independent processing elements embedded in some communications medium. There is no shared memory — all information is exchanged by explicit interprocessor communications. There must be some global addressing scheme for processing elements so that it makes sense for one processing element to transmit the address of another to a third party. Each processing element needs at least enough local memory to hold a handful of processing element addresses and small integers.

Connection graph grammars were originally devised as a tool for programming massively parallel computers, such as a connection machine [4], with processing elements only as large as the above description requires. However, connection graph grammars may work just as well for programming distributed computers possessing more powerful processing elements with more local memory, perhaps even for a collection of personal computers linked by a network. The implementation techniques outlined in this section apply equally well to any size of processing element.

It is clear how the representation of a connection graph can be distributed among the processing elements of a parallel computer: individual vertices can be held locally in processing elements, and connections can be implemented by passing around addresses.

Methods can be applied as a purely local operation. The processing elements that contain the vertices that constitute an instance of the left hand side of some method can agree to replace that subgraph with an instance of the right hand side, without bothering any other processing elements. Thus many methods can be applied in parallel throughout the machine.

Two-vertex methods are particularly easy to implement because they require the cooperation of at most two processing elements. It is hard to imagine a scheme for parallel computation in which no processing element ever has to cooperate with any other processing element, so this, in some sense, is minimal.

Any graph reduction based model can make similar claims of locality; the distinguishing feature of the connection graph model is the *connection.* Other models join objects to each other using pointer-like mechanisms, which allow an unbounded number of other objects to hold references to any given object. In a connection graph a trivalent vertex, for example, is referenced from (connected to) exactly three other places.

Synchronization between processing elements is simplified because only a fixed, small number of other processing elements have any interest in the status of a given vertex. Communications bottlenecks are eliminated because only a single copy of any given address ever exists.

At first glance, it appears that connections must be implemented using a pair of addresses, giving each terminal the address of the other terminal. In this case, splicing new structure into the connection graph after applying a method requires using the addresses stored in the terminals of the old vertices to locate the terminals that need to be reconnected to new vertices. This is clearly a general implementation technique for connection graph structure. Its communications behavior is fairly good. Only a small, fixed number of interprocessor messages need to be sent in order to execute a method.

With a little work it is possible to reduce both the number of addresses that must be stored, and the number of messages that must be sent to update the connection graph after executing a method. Static analysis of a connection graph grammar can identify connections that can be implemented using only a single address stored at one terminal addressing the other. When such a connection needs to be reconnected after a method executes, the address can simply be moved from the old terminal to the new.

This optimization can often be applied when the programmer uses connection graph structure to build conventional-looking data structures such as linked lists. If `CONS` vertices are used to build lists in the familiar way, and the programmer writes code that manipulates these lists conventionally, then it can be deduced from the resulting connection graph grammar that the connection joining the `CDR` terminal of one `CONS` vertex to the `UP` terminal of the next can be implemented by storing the address of the `UP` terminal in the `CDR` terminal. This is, of course, exactly the way people have always represented linked lists. Thus in conventional-looking cases, we can recover conventional representations.

To facilitate the identification of applicable methods, we can replace all addresses with *labeled pointers* which contain both the address of a terminal and its label. Since the left hand side of a two-vertex method is completely characterized by the labels of the two terminals joined by its single common connection, a labeled pointer allows local identification of applicable methods. The label of a terminal, and the label of the terminal it is connected to, are both made available in one place.

When a processing element discovers that a vertex in its local memory is connected to a non-local vertex in such a way that a method can be applied, the easiest way to handle the situation is to move the local vertex to the processor containing the other vertex. In the case where the optimization described above has eliminated labeled pointers that address the terminals of this local vertex, there is nothing to be done to accomplish this move other than putting the vertex representation in the mail!

Although in general things will not be this easy, at least in some cases communications costs can be reduced to this absolute minimum of a single message. It is also reassuring that at least some things the programmer can write put him in more-or-less direct contact with the primitive operations of the underlying hardware.

Translating Lisp into a Grammar

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

Variables will be a problem. Most programming languages allow use of a variable to distribute a quantity to multiple places. This is incompatible with connections, which restrict the circulation of a reference to a single location. It is not, however, unprecedented for a parallel programming language to place restrictions on variables to

avoid problems caused by widely distributed object references. In NIL [7], for example, variables are controlled so that data is *owned* by one process at a time.

Expressions

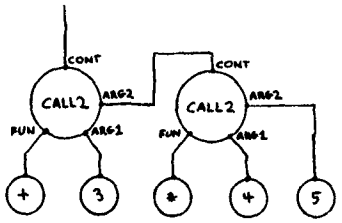

Figure 4. Graph of (+ 3 (* 4 5))

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.)

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. †

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.

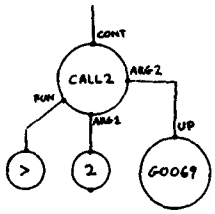

Figure 5. Graph of (> 2 (LAMBDA () (+ 3 (* 4 5))))

The connection graph for the (admittedly unlikely) expression

```
(> 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

† An interesting discussion of the relationship between expressions and the networks or graphs they notate can be found in [6]. 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.

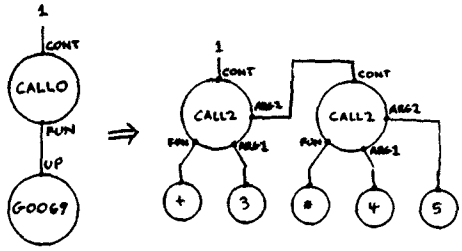

Figure 6. Method for G0069 Vertices

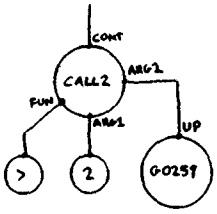

Figure 7. Graph of (> 2 (LAMBDA (Y) (+ 3 (* Y 5))))

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.

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

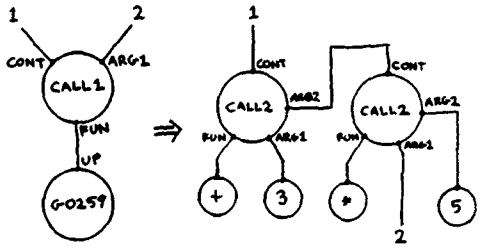

Figure 8. Method for G0259 Vertices

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

```
(> 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)))
```

Figure 9. Graph of
(LAMBDA (X) (> 2 (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)
  (> 2
     (LAMBDA (Y)
       (+ X (* Y 5)))))
```

whose rather uninteresting graph appears in figure 9.

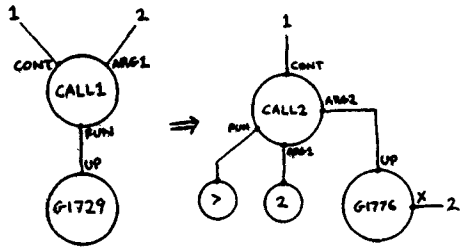

Figure 10. Method for G1729 Vertices

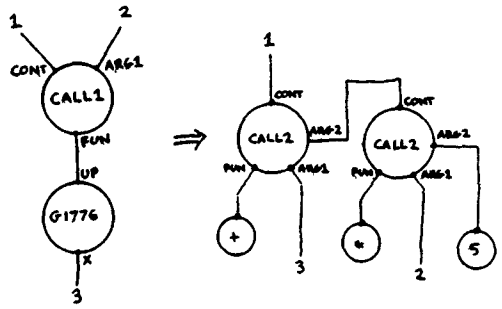

Figure 11 Method for G1776 vertices

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.

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 two-vertex 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.

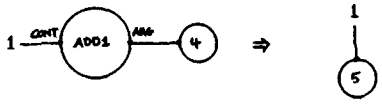

Figure 12. Method for Incrementing a 4

This suggests how a two-vertex method can also be viewed as a *message pass* [1] [9]. 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.)

When a two-vertex 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.

This remarkable aspect of programming with connection graph grammars results in the unification of several pairs of concepts that are ordinarily only near-duals. For example, if the implementation techniques for connection graphs discussed above are employed, the actions performed by the hardware when "sending a message" to a non-local object (or calling a non-local procedure) sometimes really will consist of merely sending a hardware-level message. Sometimes that message will, as expected, consist of the description of an operation and a list of arguments. The symmetry between objects and messages implies that instead the *object* may be bundled up and mailed to the location of the operation. In that case the hardware-level message will consist of a description of a type and a list of instance variables.

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 right hand side of figure 12.

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

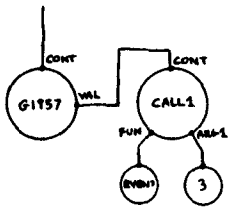

Figure 13. Graph of (IF (EVEN? 3) 4 5)

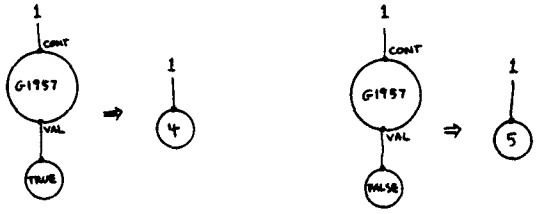

Figure 14. Methods for G1957

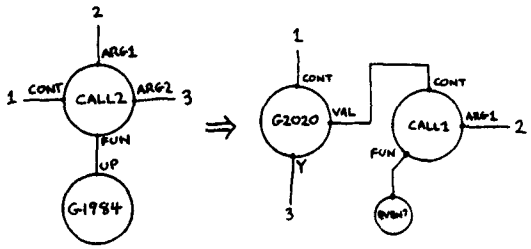

Figure 15 Method for
(LAMBDA (X Y) (IF (EVEN? X) Y (* 2 Y)))

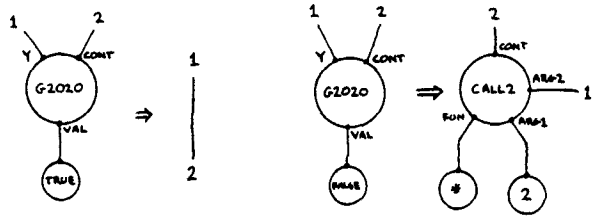

Figure 16. Methods for G2020

by returning four. The second method returns five in the case where three is odd.

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, 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.

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.

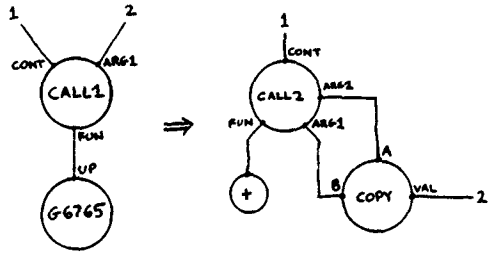

Figure 17. Method for (LAMBDA (X) (+ X X))

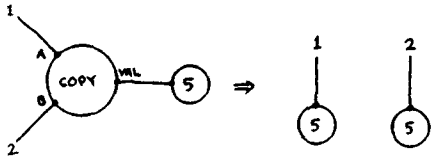

Figure 18. Method for Duplicating 5

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. Methods for the duplication of other objects can be constructed by the programmer on a type-by-type basis.

Not all types need respond to being copied by simply duplicating a vertex. An object with local state can be implemented by allowing the COPY vertices to accumulate into a fan-in tree with the state variables stored at the apex. Figure 19 shows six methods that can be used to implement an object called a CELL that responds to PUT and GET operations to modify a state variable.

The bottom four methods 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. PUT and GET vertices have a USERS terminal to hold the part of the tree they have propagated past. The top two methods are used when GET and PUT vertices encounter the CELL vertex at the apex. The GET operation returns a copy of the value stored in the cell. The PUT operation replaces the value stored in the cell and drops the old value. DROP vertices are used to dispose of unwanted connections just as COPY vertices are used to multiply access to popular ones.

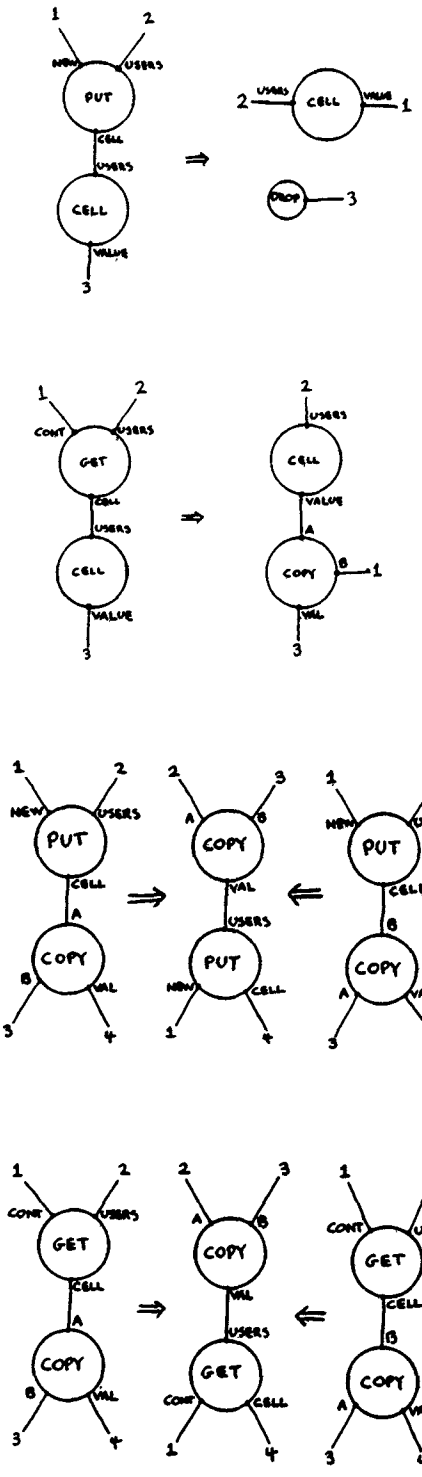

Figure 19. Methods to Implement a CELL

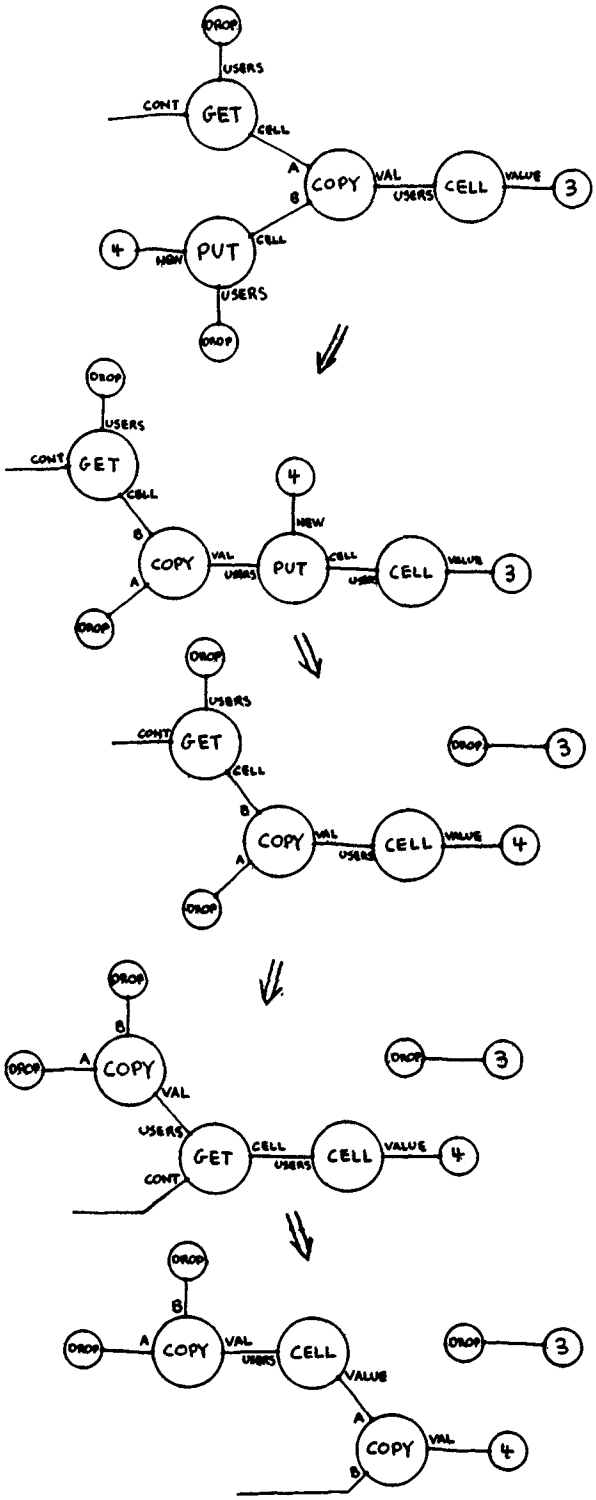

Figure 20. Using a CELL

Figure 20 demonstrates how a CELL functions. Initially 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)))
  (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 in figure 19 that can be applied to the initial graph in figure 20. 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 the second graph, where there are also two applicable methods, the method for delivering a PUT message to a CELL is chosen. In this case the choice does not change the final answer. In the next graph only the method for propagating the GET can be applied, and then in the following graph 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.)

This implementation of a CELL exhibits two phenomena well known to be associated with the notions of *object*, *state* and *side-effect* [2] [3] [5]. First, it involves a bottleneck — PUT and GET operations are forced to line up and wait their turn at the state variable. This eliminates parallelism in the connection graph grammar implementation. Second, it is non-deterministic. The order in which the methods in figure 19 are applied affects the order in which PUT and GET operations reach the apex of the tree. Thus it is difficult to reason about the outcome of a program that includes such methods.

That β-reduction is in some way incompatible with both side-effects and non-determinism is well-known. Connection graphs give us a new way to look at this incompatibility. Expressions have a basically tree-like structure; multiple occurrences of variables in an expression introduce loops into the structure, as in figure 17. When β-reduction textually substitutes expressions for variables, it eliminates the loops. Therefore, the β-reduction rule is not sound when the meaning of an expression is taken to be the connection graph it describes.

Connection graphs are a more realistic way to assign meanings to programming language expressions, because the interactions of expressions with side-effects and non-determinism are explicitly accounted for.

Implementation Status

A prototype compiler for a connection graph programming language, and a simulator, have been implemented on Symbolics Lisp Machines.

A code generator for the Thinking Machines Corporation connection machine, a fine grained, massively parallel computer [4], is currently under development.

A few small test programs have been written. We are just beginning to write our first sizable program, a parallel production system.

Conclusion

Connection graphs are well suited for implementation on at least the class of parallel computers consisting of independent, communicating processing elements. The mechanism of connections and two-vertex methods makes it easy and natural for such parallel machines to execute a connection graph grammar. With a little work, many common cases of interprocessor communication can be reduced to a single message transmission, and more efficient representations for vertices serving as conventional data structures can be deduced.

A programming language based on connection graph grammars can be constructed using two-vertex methods to implement a procedure calling and message sending mechanism. The symmetry of connections allow the notions of *object* and *message* to emerge in a new light as completely dual concepts. A difficulty arises with respect to multiple occurrences of variables in expressions, but it is shown to be merely an old adversary in new clothing.

References

1. Goldberg, Adele; and Robson, David *Smalltalk-80: The Language and its Implementation* Addison-Wesley (1983).

2. Henderson, Peter "Is It Reasonable to Implement a Complete Programming System in a Purely Functional Style?" The University of Newcastle upon Tyne Computing Laboratory (Dec. 1980).

3. Hewitt, Carl "Viewing control structures as patterns of passing messages" *AI Journal* Vol. 8 no. 3 (June 1977) 323-363.

4. Hillis, W. Daniel *The Connection Machine* MIT Press (1985).

5. Lieberman, Henry "Thinking About Lots Of Things At Once Without Getting Confused: Parallellism in Act 1" MIT AI Memo 626 (May 1981).

6. Steele, Guy Lewis Jr.; and Sussman, Gerald Jay "Constraints" MIT AI Memo 502 (November 1978).

7. Strom, Robert E.; and Yemini, Shaula "NIL: An Integrated Language and System for Distributed Programming" *SIGPLAN 1983 Symposium on Programming Language Issues in Software Systems* (June 1983).

8. Wall, David W. "Messages as Active Agents" *ACM '82 Symposium on Principles of Programming Languages* (January 1982).

9. Weinreb, Daniel; and Moon, David "Lisp Machine Manual" Symbolics Inc. (July 1981).