Linear Graph Reduction: Confronting the Cost of Naming

Alan Bawden

March 10, 1993

Abstract

Linear graph reduction is a simple computational model in which the cost of naming things is explicitly represented. The key idea is the notion of *linearity*. A name is linear if it is only used once, so with linear naming you cannot create more than one outstanding reference to an entity. As a result, linear naming is cheap to support and easy to reason about.

Programs can be translated into the linear graph reduction model such that linear names in the program are implemented directly as linear names in the model. Nonlinear names are supported by constructing them out of linear names. The translation thus exposes those places where the program uses names in expensive, nonlinear ways.

Two applications demonstrate the utility of using linear graph reduction: First, in the area of distributed computing, linear naming makes it easy to support cheap cross-network references and highly portable data structures, Linear naming also facilitates demand driven migration of tasks and data around the network without requiring explicit guidance from the programmer.

Second, linear graph reduction reveals a new characterization of the phenomenon of state. Systems in which state appears are those which depend on certain *global* system properties. State is not a localizable phenomenon, which suggests that our usual object oriented metaphor for state is flawed.

Contents

List of Figures

Chapter 1

Introduction

Linear graph reduction is a simple computational model in which the cost of naming things is explicitly represented. Names are costly because communication is costly, and names govern the way computational entities communicate. Linear graph reduction can be used to model computational systems in order to expose the places where those systems use names in expensive ways.

The key to the linear graph reduction approach to naming is the notion of *linearity*. Briefly, a name is linear if it is only used once. With nonlinear naming you can create more than one outstanding reference to an entity, but with linear naming you cannot. As a result, linear naming is cheaper to support than fully general naming, and it is also easier to reason about.

Programs can be translated into the linear graph reduction model such that linear names in the program are implemented directly as linear names in the model. Nonlinear names can be supported by constructing them out of the more primitive linear names. We can choose to view this translation as either *exposing* nonlinearity, since certain telltale structures are created by converted nonlinear names, or as *eliminating* nonlinearity, since the model itself is entirely linear. In either case, once a program has been translated into the linear graph reduction model, many problems which are exacerbated by nonlinear naming become more tractable.

To demonstrate the utility of the linear graph reduction model, I applied it it two different areas. First, I used it to solve some practical problems in the construction of a distributed programming environment. By taking advantage of the properties of linear naming I was able to build a distributed programming environment in which cross-network references are cheap, and all data structures are highly portable. Furthermore, the controlled linear naming mechanism facilitates the construction of heuristics that effectively migrate tasks and data around the network without requiring explicit guidance from the programmer.

Second, I used linear graph reduction to develop a new theoretical characterization of the notion of state. System's in which state appears stand revealed as those which depend on certain *global* properties of the system. What we normally think of as state is merely the way this property is *perceived* by the observers embedded in the system. State is not a phenomenon that can necessarily be localized, which suggests that our usual object oriented programming language metaphor for state is flawed.

The rest of this chapter introduces the notion of linearity and the linear graph reduction model and describes my two applications. Section 1.1 defines linearity and some related notions. Section 1.2 summarizes the current state of the art in distributed programming environments, and then describes how I used linear graph reduction to solve some fundamental problems in that area. Section 1.3 explains how I used linear graph reduction to investigate the origins of the phenomenon of state.

1.1 Linearity

1.1.1 What is it?

Linearity[1] is a property of names. A name is linear if it is only used once. By extension, things that use names, such as expressions and programs, are linear if all of the names they use are linear. For example

```
(define (shorter? x y)
  (< (length x) (length y)))
```

is a linear Scheme [RC92] procedure because `X` and `Y` are both used exactly once. (For simplicity, assume that global identifiers such as `LENGTH` and `<` are some kind of constant, rather than being ordinary names.)

```
(define (length l)
  (if (null? l) 0 (+ 1 (length (cdr l)))))
```

is nonlinear because `L` is used twice in the case where its value is nonempty. Note that

```
(define (f l x)
  (if (null? l) (x) (car x)))
```

is linear because `X` is only *used* once (because only one arm of a conditional or case statement is ever executed). In other words, it isn't syntactic *occurrences* of a name that matter, it is the number of times that the name must be resolved during execution.

An interpreter could be written that checks for linearity by keeping a count for each variable in every environment. Initially, each count is zero. Each time the value of a variable is retrieved, the associated count is incremented. If a

[1] Whenever I define a new term I will set it in bold type.

count for a particular variable ever exceeds one, that variable is a nonlinear name. Variables whose count never exceeds one are linear names.

Notice that this definition of linearity has little to do with the familiar meaning of "linear" from algebra.[2] The procedure

```
(define (discriminant a b c)
  (- (expt b 2) (* 4 a c)))
```

computes a nonlinear function over the numbers, but it fulfills our definition of a linear procedure because the names `A`, `B` and `C` are all used exactly once in its body. Similarly

```
(define (double x)
  (+ x x))
```

computes a linear function in the algebraic sense, while the procedure definition is nonlinear because the name `X` is used twice.

A data structure is just a bunch of names bundled together, so a data structure is linear if everything that manipulates it uses those names linearly, and if the structure itself is only manipulated once. This is really just the dual of the observation previously made about conditional and case statements. Imagine gathering up all the pieces of code that manipulate instances of the given structure and assembling them into a giant case statement that dispatches on the operation that is to be performed.[3] If we treat not only identifiers, but also accesses to structure components as names, then if that case statement is linear we will say that the original data structure was linear.

A **reference** is a low-level name—examples are memory addresses and network addresses. A **nonlinear reference** is a reference that can be freely duplicated. Most familiar references are nonlinear. In contrast, a **linear reference** is a reference that can not be duplicated. If all names in a program are linear names, then all the references manipulated at run-time can be linear references.

Linear graph reduction (which is described completely in chapter 2) is a graph reduction system in which the reduction rules are linear. In a graph reduction system edges are references, so for a graph reduction system to be linear means that the number of edges incident to a given vertex cannot change as a result of graph reduction (e.g. if a vertex is created with 3 incident edges, it will have 3 incident edges throughout its lifetime).

This is in contrast to typical graph reduction systems where the number of edges incident to a vertex can change during the vertex's lifetime. In such systems the edges are directed, and it is the number of *inbound* incident edges that can change. In particular, the number of inbound edges is normally allowed to grow without bound.

---

[2] But see the discussion in section 1.1.2.

[3] In effect, convert the program into the object oriented style of Scheme programming found in [AS85].

Since the all reduction rules in a linear graph reduction system are linear, it follows that the vertices are linear data structures in the sense defined above. We are guaranteed that at all times, all structures only have edges connecting them to a fixed number of other structures.

It is possible to translate an arbitrary computer program into a set of rules for a linear graph reduction system. (The complete translation process for a particular well-known language is described in chapter 3.) In the process of this translation, something must be done in order to implement nonlinear names that appear in the program, using only linear reductions and linear vertices. Fortunately, it proves possible to build structures that support the behavior of nonlinear names. Importantly, it proves possible to do this in a way such that the linear names are implemented *directly* in terms of edges (the native reference mechanism in the linear graph reduction model). This translation process thus "squeezes out" the nonlinearities in the original program, exposing them to view.

Another important difference between linear graph reduction and most traditional graph reduction systems is the way vertices are used. Traditionally graph reduction has been used to represent the process of normalizing an expression, and so vertices typically correspond to expressions taken from the program. In linear graph reduction, we use vertices to represent the familiar data structures found in a typical programming language implementation: records, stack frames, closures, lists, numbers, etc. The working graph directly represents the state of such an implementation captured at a particular point in time. The linear reduction rules model the changes that occur in the state of the implementation as the computation evolves.

1.1.2 Why is it called "linearity"?

It may not be immediately obvious why I have chosen to call this property "linearity". In a dictionary, the first sense given for "linear" is typically something like

> of, relating to, resembling, or having a graph that is a straight line,

which is obviously the original meaning of the word, but an additional sense is usually given as

> of the first degree with respect to one or more variables.

This second sense represents a fact that mathematicians *discovered* about functions whose graph satisfies the first sense of the word: Linear functions can be written as expressions (using only the operations $+$ and $\cdot$) in which the relevant variable is only used *once*. As a result of this discovery, the definition of linear has expanded to include this syntactic property.[4]

It is the syntactic property *alone* that I have in mind when I use the word linear. None of the other senses apply (at least in any way that I have been able to discover). Note that by itself the syntactic property is still enough to prove that compositions of linear expressions are linear. Substituting a linear expression for every occurrence of a variable in another linear expression always results in a linear result. This kind of proof works equally well for the linear functions from algebra as it does for my linear programming language expressions.

1.1.3 Why is linearity important?

The key to the importance of linearity is the observation that nonlinear naming can be used to create more than one outstanding reference to an entity—and this can be expensive. As long as an entity is only referred to through linear names, there can only be one place in the system that refers to it, but as soon as nonlinear names are used, references can multiply. Furthermore, in all traditional programming languages this is the *only* way references can multiply; unnamed entities, such as continuations and the intermediate values produced during the evaluation of expressions, are only referenced from one location throughout their lifetimes.

When references proliferate, many things can happen that would otherwise be impossible. In general, we must pay some price in order to prepare for these additional possibilities. Here are three examples that will play roles of varying importance in the rest of this dissertation:

Garbage collection. In the absence of multiple references, the storage management problem is easy to solve: The storage associated with an entity can be freed the first time it is used, because the first time must also be the last time. A garbage collector only becomes necessary when nonlinear names allow the references to entities to multiply.

Programming language implementors have traditionally taken advantage of this fact in order to stack allocate continuations. Most programming languages don't provide a mechanism for naming continuations. In those languages a continuation is only ever referenced from a single location at a time; it is either the "current" continuation, or it is referenced from some more recent continuation. Once such a continuation is used, it may be immediately discarded.

[4] The definition of linear also now includes other consequences of the original definition of the word. For example, a function T satisfying relations like

$$\begin{aligned} T(\mathbf{x}+\mathbf{y}) &= T(\mathbf{x})+T(\mathbf{y}) \\ T(a\cdot\mathbf{x}) &= a\cdot T(\mathbf{x}) \end{aligned}$$

is said to be linear—even though the domain and range of T may be such that it makes little sense to draw a graph of T in order to see if it resembles a straight line.

In programming languages that provide a mechanism for naming continuations, such as Scheme [RC92] and Common Lisp [Ste90], allocating continuations on a stack no longer works in all situations. (Common Lisp's approach to this problem is to just allocate continuations on the stack anyway, and then write the resulting limitations into the language definition.)

Distributed computing. In the absence of multiple references, there are no communications bottlenecks. If entities refer to each other using only linear names, each entity can only be known about by one other entity. When that single outstanding reference is finally used to communicate with the entity, that will necessarily be the reference's last use. So the first message received will also be the only message received. There is never any need to worry that a large number of messages may arrive in a short time and swamp the local communications facilities. Indeed, the receiver does not even need to continue to listen after the first and final message arrives.

As with garbage collection, implementors are already taking advantage of this in the case of continuations. When implementing remote procedure calls [BN84] a continuation is a link between two network locations that will eventually transport some answer back to the caller. Since a continuation can only be used once, there is never any need to devote any resources to maintaining that link after the completion of its single intended use.

This connection between nonlinear naming and the complexity of communications in a distributed environment will be central to both chapter 4 and chapter 5.

Side effects. In the absence of multiple references, side effects are impossible. Again, this results from the inability to use an entity more than once without using nonlinear naming. In this case the notion of side effect simply makes no sense unless you can interact with the same entity at least *twice*. You need to be able to perform one operation that supposedly alters the state of the entity, followed by a second operation that detects the alteration.

This observation will play a central role in chapter 6, where the phenomenon of state will be examined.

Of course we don't want to give up the language capabilities that make garbage collection necessary, or that make communications bottlenecks possible, or that allow side effects; I am not advocating a "linear style" of programming analogous to the "functional style" of programming. It is practical to program in a functional style, but linearity is too restrictive to ever inflict it directly on programmers.[5] There are times when multiple references to the same entity are exactly what the programmer needs. For example, a communications bottleneck

[5] Although in [Baw84] I did advocate exactly that. I have since come to my senses.

may be a small price to pay in return for the ability to share some valuable resource.

Fully general naming is a powerful tool. We want to be able to use that power whenever we need it. But we don't want to pay for it when we aren't using it. In fact, as many of the examples that follow will demonstrate, typical programs are largely linear. Most objects are named linearly most of the time. By implementing naming such that we only pay for nonlinear naming when we actually *use* it, we can make the common case of linear naming much cheaper. This is why it is important that the translation of a program into linear graph reduction rules implements the linear names from the program directly in terms of the linear references (edges) in the graph—if those edges are kept cheap, then linear names in the original program will be cheap.

1.1.4 Why not just use λ-calculus?

λ-calculus is a simple model of naming that has traditionally been used in the study of programming languages. Why can't the study of linear naming be carried out using λ-calculus? The problem is that λ-calculus itself exhibits the phenomenon we wish to study. λ-calculus lets you write an identifier as many times as you like. λ-calculus itself makes no distinction between linear names and nonlinear names.

For many purposes, this property of λ-calculus is an advantage. For example, a computer's memory system supports nonlinear naming—a computer's memory is willing to accept the same address over and over again—which conveniently allows a compiler to implement the high-level names from λ-calculus directly in terms of the low-level names from the memory system.

This property is what makes translating a program into "continuation-passing style" [Ste76, Ste78] an effective technique for a compiler. This translation exposes the unnamed intermediate values and continuations necessary to execute the program by turning them into explicitly named quantities. This simplifies working with the original program by reducing all reference manipulation to the named case. In particular, since the memory system supports essentially the same model of naming, compiling the continuation-passing style λ-calculus into machine language is relatively simple.

As it happens, all the names introduced when a program is translated into continuation-passing style λ-calculus are linear. But nothing in the λ-calculus explicitly represents this fact. For this reason it can be argued that the translation has actually *hidden* some important information, since some quantities that were previously only referenced through a special case linear mechanism are now referenced through the general nonlinear naming mechanism.

Compiler writers that use continuation-passing style are aware of this loss of explicit information. They generally take special pains to be sure that the compiler stack-allocates continuations (at least in the common cases). See [KKR+86] for an example. In effect they must work to recover some special case information about linearity that the translation into continuation-passing style has hidden.

The translation of a program into the linear graph reduction model is closely analogous to the translation into continuation-passing style λ-calculus. It too exposes things that were previously implicit in the program. In addition to exposing continuations and intermediate values, it also exposes nonlinear naming. This exposure comes about for analogous reasons: just as continuation-passing style λ-calculus does not support intermediate values and continuations, the linear graph reduction model doesn't support nonlinear naming.

1.2 Application to distributed computing

In order to demonstrate the power of exposing nonlinearity, I went in search of a practical problem that could be solved using this tool. Distributed computing presented itself as the obvious choice because it is an area in which naming problems are particularly acute. In a distributed environment, the user of a name and the named entity itself may be separated by the network, a situation which creates problems not present in purely local naming.

In this section, I will briefly review the state of the art in distributed computing, discuss some of the problems currently being faced, and then explain how I applied linear graph reduction to solve those problems.

1.2.1 The state of the art

The remote procedure call (RPC) is well established as a basis for distributed computing [BN84]. If what is needed is a single interaction with some remote entity, and if the nature of that interaction is known in advance, then RPC works well. RPC achieves a nice modularity by neatly aligning the network interface with the procedure call interface.

RPC provides a sufficient basis for constructing any distributed application, but performance can become a problem if the pattern of procedure calls does not represent an efficient pattern of network transmissions. Problems are caused by the fact that every RPC entails a complete network round trip. For example, reading a series of bytes is naturally expressed as a sequence of calls to a procedure that reads a buffer full of bytes, but performance will be unacceptable if each of those calls takes the time of a network round trip. A network stream (such as a TCP connection [Pos81]) will achieve the same goal with much better performance, but it is very difficult to duplicate the way a stream uses the network given only RPC. I will call this the "streaming problem".

Additional problems are caused by the fact that the result of an RPC is always returned to the source of the call. For example, suppose a task running on network node A wants to copy a block of data from node B to node C. The natural way for A to proceed is to first read in the data with one RPC to B, and then write it out with a second RPC to C. The total time spent crossing the network will be $4T$, where T is the "typical" network transit time.[6] The data itself will cross the network twice—once from B to A and again from A to C. Clearly the same job could be accomplished in $3T$ using more low-level mechanisms: first a message from A to B describes the job to be done, then a message from B to C carries the data, and finally a message from C to A announces the completion of the job. Again, it is very difficult to duplicate this behavior given only RPC. I will call this the "continuation problem".

The streaming problem can be solved in a variety of ways. For example, the RPC and stream mechanisms can be combined to allow call and return messages to be pipelined over the same communications channel. In order to take full advantage of this pipelining, the originator needs to be able to make many calls before claiming any of the returns. Some additional linguistic support is required to make such "call-streams" as neatly modular as simple RPC [LBG+88, LS88].

In many cases the streaming problem can be solved by migrating the client to the location of the data, instead of insisting that the data journey to the client. Consider the case where the individual elements of the stream are to be summed together. Instead of arranging to stream the data over the network to the client, the holder of the data is sent a description of the task to be performed; it then runs that task and returns the result to the client. This is almost like RPC, except that an arbitrary task is performed remotely, rather than selecting one from a fixed menu of exported procedures. This requires some language for describing that task to the remote site. Typical description language choices are Lisp or PostScript dialects [FE85, Par92, Sun90].

Neither of these techniques is a fully general solution to the performance problems of pure RPC. In particular, neither addresses the continuation problem, since both techniques always insist on returning answers directly to the questioner. The continuation problem could be solved by using a variant of RPC where the call message explicitly contained a continuation that said where to send the answer. In the example above, a continuation that named A as the place to send the answer would be passed in a call message from A to B; then B would send the data to C in a second (tail-recursive) call message that contained the *same* continuation; finally C would return to that continuation by sending a return message to A.

In addition to performance problems, there is another shortcoming shared by RPC and all the improvements described above: They all require the caller to specify the network node where the next step of the computation is to take place. The caller must explicitly think about where data is located. It would be much better if this was handled automatically and transparently, so that the programmer never had to think about choosing between remote and local procedure call. Instead, tasks and data would move from place to place as

[6] Each RPC call takes $2T$, T for the call to travel from caller to callee, and T for the reply to return. See [Par92] for a good presentation of the argument why ultimately T is the only time worth worrying about.

needed, in order to collect together the information needed for the computation to progress.

While RPC aligns the network interface with the procedure call interface it fails to unite the network naming system with the programming language naming system. To bring about such transparency, the references manipulated by the system must uniformly capture the network location of the referent. Such references cannot be simple pointers, since sometimes the relevant information will be local, and other times it will be remote. Using such a reference will necessarily require that these two cases be distinguished. This leads naturally to the slightly more arm's length approach to references that is characteristic of object oriented programming. An example of this is the way "ports" are used as universal references in Mach [YTR+87].

1.2.2 A unified approach

All of the various improvements on simple RPC based systems are heading in reasonable directions. Eventually these developments, and others like them, may be combined to build efficient, location transparent, distributed programming environments. But the journey down this road will be a difficult one until distributed system architects recognize that all of the problems have something in common: They are all ultimately naming issues. Specifically:

- Solutions to the streaming problem all require mobility of either tasks or data, and mobility is difficult because it puts pressure on the way entities can name each other. Anyone who has ever filed a forwarding address with the post office understands how mobility interacts badly with naming systems.

- The continuation problem is a result of the way the typical implementation of RPC fails to explicitly name continuations.

- The lack of location transparency is a deficiency in the naming scheme used for objects in the distributed environment.

In all three cases, the root of the problem is that implementors are reluctant to embrace fully general naming schemes for remotely located objects. Implementors avoid such naming because *nonlinear* naming can be expensive to support in a distributed environment, although in fact linear naming is the common case (for example, linear naming is all that is *ever* needed to solve the continuation problem) and linear naming can be supported quite cheaply.

Thus, by using linear graph reduction as the basis for a distributed programming environment, all of the problems discussed in the last section can be addressed at once. The basic strategy is to compile programs written in a familiar programming language into the linear graph reduction model, and then execute them on a distributed graph reduction engine. I have constructed such a system—its detailed description makes up the body of this dissertation. In the rest of this section I will describe how this approach solves the problems identified above.

At run-time, the vertices in the working graph will be parceled out to the various network nodes that are participants in the distributed graph reduction engine. Recall that these vertices function as stack frames (continuations), records, numbers, procedures, etc., and the edges that join them function as the references those structures make to each other. Some edges will join vertices held at the same location, and other edges will cross the network. As the working graph evolves due to the application of linear reduction rules, it will sometimes prove necessary to **migrate** groups of vertices from place to place.

1.2.3 Continuations and location transparency

The continuation problem will be solved because in the linear graph reduction model all data structures refer to each other using the same uniform mechanism (edges). So in particular, continuations will always be *explicitly* referred to by other entities. In more implementational terms, stack frames will be linked together just as any other data structures are linked together—there will be no implicit references to continuations such as "return to the next thing down on the stack" or "return to the network entity who asked you the question". A reference to a continuations that resides on a different network node will be represented in the same way as a reference to any other remote object.

Importantly, since continuations are always manipulated linearly, these references to remote continuations will always be cheap. There will never be any need to deploy the additional mechanism necessary to allow there to be multiple references to a continuation.

Location transparency will also be a consequence of using a uniform representation for all references. Since all references are the same, we can allow the names used in our programs to be implemented directly in terms of these references. This will keep the locations of all entities completely hidden from view in the programming language. In other words, a uniform representation for references allows us to align the network naming system with the programming language naming system.

Linearity facilitates these solutions by minimizing the cost of adopting a uniform representation for references. We *could* have achieved the same results using nonlinear references—at a price. As long as the nonlinear references to local and remote entities had the same representation, and as long as continuations were referenced explicitly, we would have solved the continuation problem and achieved location transparency. These nonlinear references would be difficult, but not impossible, to maintain across the network.

However, as I will demonstrate in (almost painful) detail in chapter 4, cross-network linear references are very simple and easy to maintain. This is a direct result of the fact that linear references cannot be duplicated. Supporting linear references from remote locations to a local entity is easy because you only need to worry about the whereabouts of *one* outstanding reference. If that reference ever becomes local, then all network resources devoted to maintaining the reference can be easily discarded. If the referenced entity needs to move to a new location, only that single reference needs to be tracked down and informed of the new location.

1.2.4 The streaming problem, demand migration and tasks

Cheap linear references also play a role in solving the streaming problem. Recall that the streaming problem is an instance of the more general problem of mobility: a task is separated from the data it wants to manipulate next, and so either the task or the data must be migrated. Since both task structures and data structures contain references to, and are referenced by, other entities, using linear references makes both tasks and data easier to migrate.

But that doesn't directly address the issue of *which* structure to migrate. Should we migrate the data to the task, or the task to the data? Nor does it address the much harder problem of selecting what to migrate and what to leave behind. If we decide to migrate a list, should we send the elements of that list along for the ride? If we send a stack frame, what structures directly referenced from that stack frame should accompany it? Should we also include the next stack frame down?

These wouldn't be problems if we simply reverted to the current state of affairs, where the programmer must explicitly specify everything about a migration. But having successfully hidden the network from the programming language level so far, it would be a shame to surrender at this point. Fortunately, linearity can be used in a different way to solve this problem as well. The solution is closely analogous to the way demand paging works to implement virtual memory. When it becomes clear that a migration must take place, we will handle this "fault" by using some simple heuristics to select a set of vertices to migrate. Just as demand paging satisfies a page fault by bringing more words into memory than are strictly necessary to relieve the fault, we will migrate additional vertices beyond those that are immediately needed. This process will be called **demand migration**.

In demand paging, the heuristic is to read in words adjacent to the one that caused the fault. This works well because of the property commonly called "locality": addresses referenced in the recent past tend to be close to addresses that will be referenced in the near future. Using exclusively linear structures enables us to make analogous predictions about what structures will need to be close to each other in the near future, so we can avoid future separate migrations ("faults") by migrating those structures together.

The migration heuristics work by combining the limited and controlled nature of linear references with some simple compile-time analysis of the set of linear graph reduction rules. Precisely how this works will be explained in chapter 5, but the basic contribution made by linearity is to make it easy to know when one entity (A) makes the *only* reference to another entity (B). In such a case, if A is definitely going to be migrated, then it is likely that migrating B is a good choice. This observation simplifies the problem enough that a few heuristics are able to make migration choices that are close to optimal.

In fact, the heuristics are so good that they are able to reconstruct the intuitive notion of a "process" or "task", even though the linear graph reduction model itself supports no such notion. Intuitively a task is some locus of ongoing activity along with all of the data structures that are directly related to supporting that activity. In other words, a task is exactly the right thing to treat as a unit when migrating things around a network.

In order to take full advantage of programming in a distributed environment, there needs to be some mechanism for getting more than one task running at the same time. For this purpose I have chosen the `FUTURE` construct as found in MultiLisp [Hal84]. (Futures have a particularly elegant implementation in the linear graph reduction model.) We will see that the migration heuristics work well at spotting the linear graph structure that properly "belongs" to the various tasks created when using futures, and so the system will be able to migrate those tasks separately around the network. All this from a model that has no native notion of task to begin with.

1.2.5 Results

In a distributed environment, parts of an evolving computation can become separated from each other. A fundamental problem is bringing these parts back together again when the appropriate time arrives. This is precisely what naming is *for*. We name things that we are not *currently* interacting with so that we may track them down *later* and interact with them then. Thus, we should expect that something that claims to be an improved way to think about naming should find application in distributed computing.

Linear graph reduction passes this test. My implementation demonstrates how proper attention to linearity can make cross-network references cheap, can support highly mobile data structures, and can facilitate heuristics to effectively migrate tasks on demand without explicit guidance from the programmer.

1.3 Application to the problem of state

As a second demonstration of linear graph reduction, I will present (in chapter 6) an examination of the phenomenon of *state*. In this context, linearity yields an important insight rather than immediately practical benefits.

Most treatments of state start with some primitive elements that support state and then study mechanisms used to combine those elements in ways that keep the state well behaved [GL86, SRI91]. My approach differs in that I will be creating entities that support state out of nothing. Then a careful examination of the conditions necessary for state to emerge will reveal the essence of what state is really all about. Understanding that essence will reveal why the usual approaches to state sometimes go awry. The rest of this section outlines my approach to state and summarizes the resulting insight.

The first problem we will face is the lack of a satisfactory preexisting definition for "state". Typically state is taken to be a property that can be localized to specific entities—some entities have state and other entities are "stateless"—but as our goal, in part, is to discover *whether or not* state can be localized, we cannot adopt such a definition. Instead, state will not be defined at all initially, and we will rely on an approach that pays careful attention to the *perceptions* of the entities in a computational system. That is, we will examine systems in which some components *perceive* state in other components. Later, armed with a successfully characterized those systems, we will be able to supply a definition for "state". (This entire enterprise can be viewed as a search for this definition.)

This is a subtle approach, and it is interesting to consider how it is that the linear graph reduction model suggested it. Using linear references makes it much clearer where the boundary lies between a base-level computation and a meta-level computation that manipulates or reasons about it. A linear reference to an entity in the base-level computation can be held by another base-level entity, but for that same reference to be held by some meta-level entity would imply (because of linearity) that *no* base-level entity can be holding that reference. This clearly reveals a confusion of levels and suggests that some other *independent* reference mechanism is required for the meta-level to name the entities in the base-level. Since state is perceived *through* the references to an entity, this suggests that state as perceived from the meta-level may not be the same as state perceived from within the base-level. We restrict our attentions to base-level perceptions because this is the viewpoint that our programs actually possess.

At this point in the dissertation we will have seen many examples of linear graph reduction systems in which various components perceive state. We will, for example, have seen the Scheme primitive `SET-CAR!` implemented in linear graph reduction terms. So by then the reader will have no trouble seeing that *all* such systems share two common characteristics:

- They are all nondeterministic. That is, they all require sets of graph reduction rules in which the outcome of a computation can depend on the order in which rules are applied.

- They all require linear graph structure that contains cycles.

These observations, especially the observation of cycles, will motivate us to take an apparent digression into the formal mathematical properties of linear graph structure. In particular we will study homomorphisms between linear graphs as a tool that will enable us to examine local vs. global properties.

Then we will prove some theorems about the actions of linear graph reduction rule sets when applied to graphs that have the same local structure but different global structure.

At this point I will pull a rabbit out of my hat, for it turns out that linear graph reduction systems that are sensitive to the global structure of the system (in a certain well-defined way) *must* be nondeterministic and *must* exhibit cycles—precisely the symptoms observed empirically in systems that experience state. This implies that what is really going on when the components of a system experience state is that the system as a whole has this kind of global sensitivity. I claim that this is how state should be defined. What we normally think of as state is merely the expression of a global property of a system in the perceptions of the observers embedded in that system.

This insight can be applied to a number of different problems to suggest new approaches to their solution. Perhaps the most promising is in the area of programming language design. Characterizing state in these global terms reveals that the usual programming language metaphor for managing state, the notion of a mutable "objects", contains a flaw—state is not something that can necessarily be assigned a single location, state is more of a global tension between a number of entities. Given this clearer understanding of what state is all about, we can hope to develop better programming language constructs that more accurately match state's true nature.

No such programming language constructs, or other practical derivatives of this insight into state, have been developed. So far this new perspective remains a surprising and subtle curiosity. This is in contrast to the distributed linear graph reduction engine, which is a quite straightforward demonstration of the practical advantages of thinking about linearity. I believe, however, that in the long run this characterization of the phenomenon of state will prove to be the more important result.

1.4 Outline

Chapter 2 describes the linear graph reduction model itself, and describes the general principles used whenever linear graph reduction is used as a model to expose linearity. It also introduces the notation used throughout the rest of the dissertation.

Chapter 3 is the first of three chapters that describe the distributed programming environment. It describes the compiler used to translate a Scheme program into linear graph reduction rules. This chapter should leave the reader with a solid understanding of how linear graph reduction can support all familiar programming language constructs.

Chapter 4 describes the basic network protocol used to support the distributed linear graph reduction engine. In this chapter we will see the first example where linearity has a tangible benefit. Because it only supports linear naming, the protocol is quite simple and light-weight. This chapter can be skimmed by people who are scared of networks.

Chapter 5 describes the algorithms and heuristics used by the individual agents that make up the distributed reduction engine. In a second example of a benefit derived from linearity, these agents use some linearity-based heuristics to further improve the communications behavior of the execution of the program.

In chapter 6 linearity is used to examine the phenomenon of state. It develops an interesting and potentially useful characterization of systems in which state occurs. This chapter only requires an understanding of the the material up through chapter 3. This chapter can be skipped by people who are scared of higher mathematics.

Chapter 7 puts this work in context by comparing it to other related (and not so related) work on naming, programming languages, distributed computing, graph reduction, and state. This chapter also discusses future directions that research into linearity can explore, and it summarizes what I feel are the most important contributions of this work.

Chapter 2

Linear Graph Reduction

This chapter introduces the abstract linear graph reduction model. The first section describes a static structure called a **linear graph**. The second section describes the **linear graph grammar**, which is a collection of reduction rules that can be applied to a linear graph to cause it to evolve. The third section describes a simple and convenient textual representation for these structures that will be used throughout the rest of this dissertation. The final section is a brief introduction to the ways in which linear graph reduction can be applied to real problems to expose linearity.

2.1 Linear graphs

Intuitively, a linear 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 linear graph differs from a circuit chiefly in that we restrict the way terminals can be connected. In a linear 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 linear graph all the terminals of any particular type of gadget must be distinguishable.

Some convenient terminology: The gadgets in a linear 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

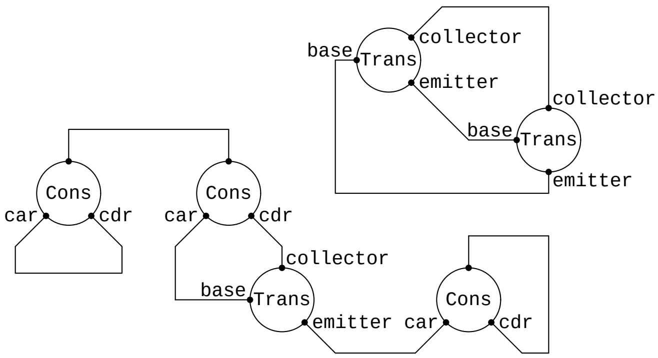

Figure 2.1: A Linear Graph

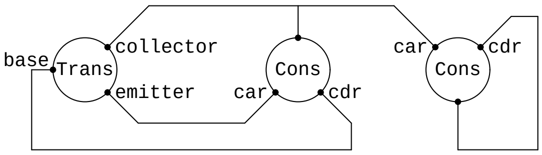

Figure 2.2: A Nonlinear Graph

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 linear 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 linear 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 2.1 shows a linear graph over the two types `Transistor` and `Cons`, where type `Cons` has the three associated labels "`car`", "`cdr`", and "" (the empty string). This example is a single linear graph—a linear graph may consist of several disconnected components.

Figure 2.2 is *not* an example of a linear graph; the "" 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 what makes the graph "linear".

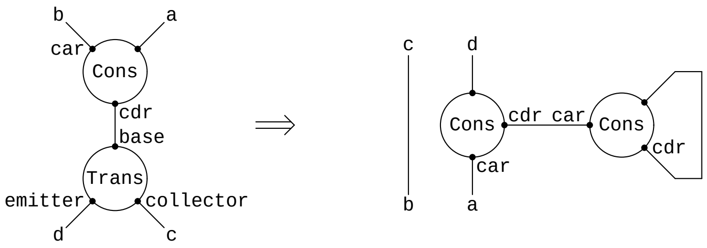

Figure 2.3: A Method

2.2 Linear graph grammars

A **linear graph grammar** is a collection of reduction rules called **methods**. Each method describes how to replace a certain kind of subgraph with a different subgraph. If the linear graphs over some set of types are analogous to the strings over some alphabet, then a linear 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 linear graph grammar the notion of replacement must be treated more carefully. Figure 2.3 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 labeled to indicate how this reconnection is to be done.

For example, when applying the method in figure 2.3, 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 linear graph that was connected to the "" terminal of the old `Cons` vertex is reconnected to the `car` terminal of the first new `Cons` vertex, as shown by the loose ends labeled `a`. The terminal that was connected to the `emitter` terminal of the old `Transistor` vertex is reconnected to the "" terminal of the same new `Cons` vertex, as shown by the loose ends labeled `d`. 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 labeled `b` and `c`.

A subgraph that matches the left hand side of some method is called a **redex**, and the act of applying a method is called **reduction**. It should be emphasized that when a reduction occurs, the redex is discarded. In this aspect linear 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 and define the "language" generated by a given linear graph grammar as the set of linear graphs that can be generated by starting with a graph consisting of a single vertex of some initial type. There might be interesting results to be proved, for example, about what kind of linear graphs can be generated using only context sensitive linear graph grammars, where that notion is suitably defined.

But none of these potentially interesting paths will be explored in what follows. In using linear graph grammars as a model of computation we will have no need of the notion of a generated language. Furthermore, only one kind of method will appear in the linear graph grammars described below: methods whose left hand side consists of exactly two vertices joined by a single connection. Figure 2.3 is an example of such a **binary method**. There will never be any need for a method whose left hand side contains a cycle, more than one connected component, or other than two vertices.

2.3 Graph expressions

Graph expressions are a textual representation for linear graphs that are more convenient than the pictures we have used so far. As an example, here is the graph expression for the graph in figure 2.1:

```
(graph ()
  (<Cons> 0 car:1 cdr:1)
  (<Cons> 0 car:2 cdr:3)
  (<Transistor> collector:3 base:2 emitter:4)
  (<Cons> 5 car:4 cdr:5)
  (<Transistor> collector:6 base:7 emitter:8)
  (<Transistor> collector:6 base:8 emitter:7))
```

A graph expression superficially resembles the S-expression representation used for λ-abstractions. A graph expression is written as a parenthesized list whose first element is the word "`graph`". The second element is an ordered list of loose ends, and the rest of the elements represent the vertices. Each vertex is written as a parenthesized list whose first element is the type of the vertex, and rest of whose elements are the vertex's terminals. A type is written by enclosing its name between angle brackets. Each terminal is written as a label and an **index** separated by a colon. (If the label is the empty string, then the colon is omitted.)

The indices specify how the vertices are connected together. Each index corresponds to one connection. An index thus appears in the graph expression exactly *twice*, once for each of the two terminals the connection joins. If one or both of the ends of a connection are loose ends, then the index appears once or twice in the list of loose ends at the beginning. Indices are typically small non-negative integers, but the values chosen are arbitrary. Indices are only meaningful within a single graph expression.

In order to generate the graph expression from a linear graph, start by picking an index for each connection. Each of the two ends of a connection is either joined to a terminal, or it is dangling loose. For each connection attach its index to the two things at its ends. Every terminal and every loose end now has an attached index, and every index has been attached to exactly two things. The graph expression is constructed by (1) reading off the indices attached to the loose ends, and (2) visiting each vertex and reading off the indices attached to its terminals.

In order to generate a picture from a graph expression, start by drawing all the vertices listed, without drawing any connections between them. Pick a point for each loose end indicated in the graph expression. Now consider the indices one at a time. Each index will occur exactly twice in the graph expression. Draw a connection between the corresponding two points.

A method is written as an ordered pair of graph expressions. The two graph expressions must have the same number of loose ends. The ordering of the loose ends determines the correspondence that is to be used when the method is applied. Thus the method in figure 2.3 is written as follows:

```
(graph (0 1 2 3)
  (<Cons> 0 car:1 cdr:4)
  (<Transistor> collector:2 base:4 emitter:3))
(graph (0 1 1 2)
  (<Cons> 2 car:0 cdr:3)
  (<Cons> 4 car:3 cdr:4))
```

With a little practice, it isn't difficult to read graph expressions directly, but initially the reader may find it helpful to draw the graphs by hand to help visualize them.

2.4 Modeling real systems

Linear graph reduction can be used to model real computational systems by using linear graphs to represent data structures, and using methods to encode algorithms. The precise details of how the modeling is done may vary from application to application, but there are some common ideas that are set forth in this section.

The basic idea is to translate the program to be executed into a linear graph grammar—a collection of types and methods. This is different from the original combinator model [Tur79] where the program becomes a *graph*, and the set of vertex types and reduction rules are fixed from the start.[1] All of the properties of the program must be encoded somehow in a set of methods.

The fact that applicable methods are selected nondeterministically means that ordering constraints in the program must be satisfied by ensuring that methods only become applicable in the desired order. Typically this is done by having one method introduce a certain vertex type into the graph that appears in the left hand side of another method. This ensures that the second method can only run *after* the first method, since the first method must produce a vertex of that type before the second method can consume it. In effect, the program counter is represented as the presence of one of a selected set of vertex types in the working graph.

A more important problem than representing the sequentiality required in the input program is representing the nonlinearity it requires. Linear graph reduction itself is linear—the connections that hold the graph together are constrained to behave in a linear fashion (a connection joins a terminal to *exactly* one other terminal), and methods are defined to preserve that property. Thus, when data structures and algorithms are built from linear graphs and methods, the nonlinear parts of the algorithms will have to be expressed in terms of more primitive linear elements.

2.4.1 Representing linear structures

Representing *linear* data structures is generally quite straightforward. The basic idea is that objects traditionally represented using N-component record structures are instead represented using vertices of valence $N+1$. The extra terminal, called a **handle**, is used as a place for other vertices to connect to. So, for example, where a pointer would be installed in the jth component of record structure A pointing to record structure B, a connection is made from the jth terminal of vertex A to the handle terminal of vertex B. (Handle terminals are usually given the empty string as their label.)

All attributes of an object other than its references to other objects are encoded using the type of the vertex. Integers, for example, make *no* references to other objects; their only attribute is their numerical magnitude. Thus an integer is represented as a univalent vertex, and each integer is given its own unique type.[2]

Representing nonlinear data structures requires some additional mechanism. This is, of course, the goal of this entire exercise. Because we are using a

[1] In some more evolved combinator implementations the compiler does introduce new vertex types and reduction rules [Pey87].

[2] This may seem like an excessive number of types to introduce in such an offhand manner, but this profusion of types is only for the convenience of the abstract model, and does not translate into any inefficiencies in any actual implementation. Note that the notion of vertex type has nothing to do with the notion of data type or variable type encountered in typed programming languages. That notion of type is actually more closely related to the labels placed on the terminals in a linear graph.

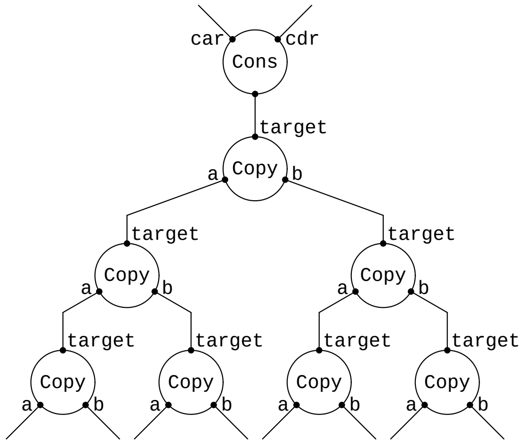

Figure 2.4: A `Copy` tree

representation that supports linear data structures directly, but that allows nonlinear structures to be built explicitly when necessary, the nonlinearities in the original program will be exposed.

2.4.2 Representing nonlinear structures

The exact mechanism used to represent nonlinear structures varies, but it is always based on the vertex types `Copy` and `Drop`. When the program requires a reference to an object to be duplicated, a trivalent `Copy` vertex is attached to the object's handle terminal by its `target` terminal. The other two terminals of the `Copy` vertex, labeled `a` and `b`, are then available to be used in place of the original handle terminal. As more and more copies of the original reference are made, more and more `Copy` vertices accumulate in a tree between the users and the object they wish to reference. Figure 2.4 shows an example of such a tree.

Similarly, when the program wishes to discard a reference, it attaches that reference to a univalent `Drop` vertex. To make `Drop` work properly in combination with `Copy`, the following two methods are always assumed whenever both types are in use:

```
(graph (0 1)
  (<Drop> 2)
  (<Copy> target:0 a:2 b:1))
(graph (0 0))

(graph (0 1)
  (<Drop> 2)
  (<Copy> target:0 a:1 b:2))
(graph (0 0))
```

These methods ensure that if a reference is duplicated, and then one of the copies is discarded, the graph will return to its original state. (Methods for balancing the tree of `Copy` vertices are not needed.)

Since the users of an object are no longer directly connected to it, it is more difficult for them to interact with the object—some additional protocol must be employed. There are basically two techniques that are used, depending on whether the object or the users take responsibility for handling the situation. The object at the apex of the tree can take responsibility by reacting to the `Copy` vertex by somehow making a suitable copy of itself. For example, numbers will usually react by duplicating themselves as follows:

```
(graph (0 1)
  (<Copy> target:2 a:0 b:1)
  (<Number 9> 2))
(graph (0 1)
  (<Number 9> 0)
  (<Number 9> 1))
```

Alternatively, the users at the fringe of the tree can take responsibility by climbing up the `Copy` tree to the object at the apex, using methods similar to these two:

```
(graph (0 1 2 3)
  (<Car> target:4 tail:2 cont:3)
  (<Copy> target:0 a:4 b:1))
(graph (0 1 2 3)
  (<Car> target:0 tail:4 cont:3)
  (<Copy> target:4 a:2 b:1))

(graph (0 1 2 3)
  (<Car> target:4 tail:2 cont:3)
  (<Copy> target:0 a:1 b:4))
(graph (0 1 2 3)
  (<Car> target:0 tail:4 cont:3)
  (<Copy> target:4 a:1 b:2))
```

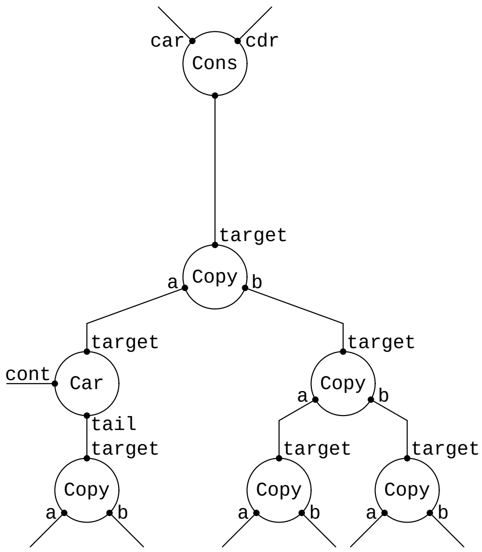

Figure 2.5: A `Car` message in transit

These methods permit a `Car` vertex to climb up through a tree of `Copy` vertices to reach the object at the apex. Figure 2.5 shows a `Car` vertex in the process of making such a journey; after the first method above is applied, the `Car` vertex arrives at the apex of the tree, as shown in figure 2.6. (Note how the `cont` terminal is being carried along for the ride.)

Vertex types, such as `Car` in this example, which climb through `Copy` trees in this fashion are called **messages**. The two terminals of a message that are used to perform this climb are customarily labeled `target` and `tail`, just as they are above.

Usually some mix of these two techniques is used to cope with the `Copy` trees introduced by nonlinearities. Small objects, such as numbers, will duplicate themselves, while large objects will respond to messages that know how to tree-climb.

2.4.3 Representing objects with state

Messages that tree-climb are also the usual technique used for supporting objects that have state. For example, the `Car` message just introduced may interact with a `Cons` vertex using a method like the following:

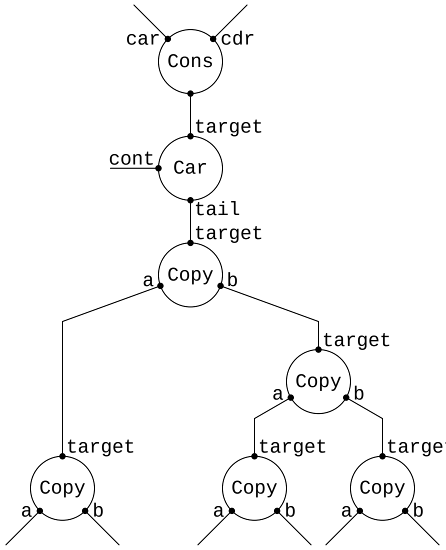

Figure 2.6: The Car message arrives

```
(graph (0 1 2 3)
  (<Car> target:4 tail:2 cont:3)
  (<Cons> 4 car:0 cdr:1))
(graph (0 1 2 3)
  (<Cons> 2 car:4 cdr:1)
  (<Copy> target:0 a:3 b:4))
```

This method returns a copy of the `car` of the `Cons` to the continuation object (presumed to be attached to the `cont` terminal of the `Car` vertex), and also replaces the `Cons` vertex at the apex of the tree. A similar tree-climbing `Set Car` message allows a `Cons` to be mutated using this method:

```
(graph (0 1 2 3 4)
  (<Set Car> target:5 tail:2 cont:3 new:4)
  (<Cons> 5 car:0 cdr:1))
(graph (0 1 2 0 4)
  (<Cons> 2 car:4 cdr:1))
```

This method returns the *old* `car` of the `Cons` to the continuation object, and replaces the `Cons` vertex at the apex of the tree with one whose `car` terminal is connected to the replacement value (previously attached to the `new` terminal of the `Set Car` vertex).

Both of the last two methods use a particular protocol for returning a value to the continuation object. The choice of continuation protocol is important, and will be examined in more detail in later chapters; for the moment I have chosen to employ a simple protocol where a continuation is simply attached directly to the return value.

It is clear from this example that tree-climbing messages are a fully general protocol that allows the graph structure at the apex of a `Copy` tree to react in an *arbitrary* way to the messages that are funneled up to it. Just how arbitrary this behavior can be is illustrated by the following example:

```
(graph (0 1 2 3)
  (<Abracadabra> target:4 tail:2 cont:3)
  (<Cons> 4 car:0 cdr:1))
(graph (0 1 2 3)
  (<Drop> 0)
  (<Drop> 1)
  (<Nil> 2)
  (<True> 3))
```

Here we have an **Abracadabra** message that changes a **Cons** into an empty list! This is not something that can occur in an ordinary Lisp implementation,[3] where the type of an object is not something that can be mutated.

Notice that the `Copy` tree serves to serialize the messages that arrive at the apex. Choices made about which methods to apply while messages are climbing up the tree will determine the order that those messages arrive at the top.

It may seem surprising that you can create side effects with something as simple as linear graph reduction. Other forms of graph reduction are not generally credited with this ability. Actually, this has little to do with the nature of graph reduction itself; it is an artifact of how graph reduction is generally viewed as an efficient way to perform expression reduction. Here, we are taking a different view, where graph reduction is used to model the data structures that implementors build to support programming languages, rather than modeling the expressions that theorists use to think about programming languages.

2.4.4 Representing input/output behavior

At some level, something other than graph reduction must take place in order to cause characters to appear on a terminal. Fortunately, it is possible to avoid confronting such low-level issues directly. Instead, we simply imagine that some *other* part of the graph is responsible for performing primitive I/O, and concentrate on how we interact with *it*. (This is similar to the way most modern programming languages support I/O facilities. Instead of providing I/O directly, they specify an interface to a library of I/O procedures.)

[3] Although beginning Lisp students sometimes wish that it could—especially when they are first introduced to the `DELQ` function.

This approach to I/O has an interesting consequence for the linear graph reduction model. It means that subgraphs can be discarded if they get disconnected from the parts of the graph that are connected to the I/O facilities. The reason for this is that no matter what methods may apply to such a subgraph, it can never become reconnected,[4] and thus it is unable to effect the output of the computation.

Since disconnected components can be discarded from the working graph at run-time, they can also be discarded from the right hand sides of methods to avoid introducing them in the first place. When we compile linear graph grammars for execution on real hardware, this will be an important optimization. (See section 3.3.) Discarding disconnected components is similar in spirit to conventional garbage collection.

This completes the introduction to linear graph reduction. A thorough familiarity with linear graph reduction and especially with the modeling techniques introduced in this section will be assumed in the following chapters. I therefore encourage you to take some time out now to *play* with some simple grammars and with modeling various kinds of data structures.

You should resist the temptation to make up new kinds of methods. Only binary methods will be needed in the following chapters.

A good problem is to design a collection of methods that implement a combinator machine. The challenge comes from the fact that both the S and K combinators are nonlinear:

$$Sxyz \Rightarrow xz(yz)$$

duplicates the value z, and in

$$Kxy \Rightarrow x$$

the value y is discarded. Using `Copy` trees, as described above, is the obvious solution, but other solutions are also possible.

[4] This is a consequence of the fact that the left hand side of a binary method is connected. If methods with disconnected left hand sides were allowed, then reconnection, and all kinds of other non-local effects, are a possibility.

Chapter 3

Compiling Scheme

In the next three chapters I will present a practical implementation of a distributed linear graph reduction engine. This system allows a programmer to program in a standard sequential programming language (Scheme [RC92]), which is then compiled into a linear graph grammar, and executed collectively by several processors communicating over a network. The goal of building this system was to demonstrate how linear graph reduction keeps the implementation simple and efficient, and allows some powerful new techniques to be applied to the problems of distributed execution.

This chapter explains how Scheme is compiled into a linear graph grammar. It is about how to correctly capture the semantics of standard Scheme using linear graph structure and methods. It is *not* about how to design a programming language that takes full advantage of being based on linear graph reduction.

Chapter 4 describes the fundamental network protocol and data structures used to maintain *distributed* linear graph structure. The fact that connections are more constrained than full-fledged pointers keeps this protocol cheap and fast.

Chapter 5, describes the distributed linear graph reduction engine. It describes the behavior of the individual agents that hold the vertices that make up the working graph. Agents exchange vertices through the network and they apply methods to the subgraphs composed of the vertices they currently possess. Agents decide which vertices to migrate, when to migrate them, and who to migrate them to, by using a simple model of the expense of interagent connections.

Chapter 4 and chapter 5 are kept distinct because they describe two different ways in which linearity is important to the system. Chapter 4 demonstrates how data structures that only name each other linearly are cheap to support in a distributed environment, where traditional non-linear naming mechanisms would be burdensome. Chapter 5 shows how linearity can be further exploited to make decisions about migrating tasks and data in order to make distributed execution more efficient.

Scheme was chosen as the source language as an example of a typical sequential programming language. Scheme is a lexically scoped, call-by-value dialect of Lisp. Scheme is described in [RC92]. Like most other Lisp dialects Scheme is "mostly functional", which means that Scheme programs are mostly written in the functional subset of the language—only occasionally are imperative features used. Scheme is also deterministic; in particular it makes no concessions to the possibilities of concurrent execution.

In theory FORTRAN or C could have served instead of Scheme (we will not be restricted to the functional subset of Scheme) but since it turns out that functional constructs are much more straightforward to compile, a language that is mostly functional makes a better choice for expository purposes.

Sticking to a purely sequential deterministic programming language fails to demonstrate a number of interesting things about distributed linear graph reduction. A purely sequential program calls for only a single action to take place at a time; a sequential program describes only a single task. Thus, even in a distributed environment there will only be a single locus of activity—although that locus may move if the task must migrate in order to access some remote resource. In order to get things executing concurrently, we need to extend Scheme with some kind of parallel programming construct. Fortunately, there is a fairly well-known Scheme extension, the `FUTURE` special form [Lie81, Hal84, Mil87], that can meet our needs.

Of course I didn't have the time to make this a truly complete Scheme implementation. There are many data types, procedures and special forms missing. In principle these missing pieces can all be supplied. One missing piece that you should not be fooled into thinking is a significant omission is the `SET!` special form. Using the technique of **assignment conversion** [KKR+86] any program that makes use of assignment statements can be transformed into one that uses mutable objects, such as `CONS`-cells, instead. Since I *have* implemented mutable `CONS`-cells, I could have done assignment conversion as well. However, this would probably double the size of the current small compiler, so in the interest of simplicity I left it out.

3.1 Vertex types for the Scheme run-time

This section describes the vertex types used by the Scheme run-time world. Many of the techniques used by these types were introduced in section 2.4.

The vertex types described in this section can be classified according to the hierarchy shown in figure 3.1. This hierarchy will double as the outline for the rest of this section.

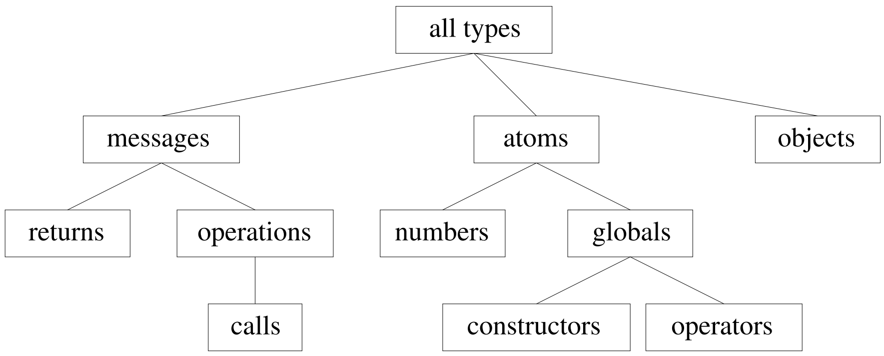

Figure 3.1: Run-time vertex types

3.1.1 Objects

An **object** is a vertex used in the way ordinary data structures are used in a traditional Scheme implementation. Recall the technique introduced in section 2.4: an N-component aggregate data structure becomes an $N+1$ terminal vertex, where the extra terminal serves as a handle for users of the object to connect to. The handle terminal is always given the empty string as its label.

The following object types are built in to the Scheme run-time system:

Type	Labels
`Cons`	“”, “`car`”, “`cdr`”
`Continuation`	“”, “`cont`”
`Sum`	“”, “`left`”, “`right`”
`Difference`	“”, “`left`”, “`right`”
`Product`	“”, “`left`”, “`right`”
`Equal?`	“”, “`left`”, “`right`”
`Less?`	“”, “`left`”, “`right`”
`Greater?`	“”, “`left`”, “`right`”

In addition, the compiler makes more object types as a result of translating the user's program (see section 3.2).

All object types have a method for the case where a `Drop` vertex is connected to the object's handle. For example, here is the method for the `Cons` type:

```
(graph (0 1)
  (<Cons> 2 car:0 cdr:1)
  (<Drop> 2))
(graph (0 1)
  (<Drop> 1)
  (<Drop> 0))
```

Similar methods exist for all of the other types in the table above, as well as for all the compiler-produced object types.

Object types specifically do *not* have a method for when a `Copy` vertex is connected to the object's handle. Instead, the `Copy` vertices are allowed accumulate into a tree, with the object vertex at the apex. The message climbing technique described in section 2.4.2 is used to communicate with such vertices. (See also the message types described in the next section.)

The type `Cons` is used to implement Scheme `CONS`-cells. Additional methods, described below, support the Scheme procedures `CAR`, `CDR`, `SET-CAR!`, `SET-CDR!`, and `NULL?`.

The type `Continuation` is used by the `CALL-WITH-CURRENT-CONTINUATION` procedure for the continuations that it creates. The `cont` terminal of a `Continuation` is connected to a "raw" continuation of the kind generally produced by the compiler. This is explained in more detail in section 3.1.3.1.

The types `Sum`, `Difference`, `Product`, `Equal?`, `Less?`, and `Greater?` are used to implement arithmetic. In order to perform addition, subtraction, multiplication or comparison, one connects the values to be added, subtracted, multiplied or compared to the `left` and `right` terminals of a `Sum`, `Difference`, etc.

Of course, simply building a tree that represents the arithmetic to be performed isn't very satisfactory when one wants to compute an actual numeric answer, so the Scheme run-time behaves as if methods such as

```
(graph (0 1)
  (<Sum> 0 left:2 right:1)
  (<Number x> 2))
(graph (0 1)
  (<Sum x> 0 right:1))
```

and

```
(graph (0)
  (<Sum x> 0 right:1)
  (<Number y> 1))
(graph (0)
  (<Number (x + y)> 0))
```

were defined for all numbers x and y. These methods make trees of arithmetic operators and numbers simplify into numbers (or to the boolean values `True` and `False` in the case of the comparison predicates).

This simplification proceeds in parallel with the evaluation of the rest of the program. This parallelism must not be exposed to the programmer if the sequential semantics of Scheme are to be faithfully reproduced, so the compiler will be careful to construct vertices such as `Sum` at exactly the moment when the sequential semantics calls for an addition to take place.

A useful way to think about this is to imagine that a vertex of a type like `Sum` is *itself* a number, so the act of constructing such a vertex is equivalent to performing an addition. The simplification process merely converts one kind of number into another more convenient kind of number.

This arithmetic-by-construction-and-simplification scheme is by no means the only way arithmetic could have been implemented. Its advantage is that the compiler is very good at compiling calls to constructors (because ultimately it must express *everything* in terms of constructing graph structure), so treating calls to `+` in the same way as calls to `CONS` results in much simpler generated code (i.e. fewer methods). It has the practical disadvantage that the arithmetic simplification process may not proceed quicky enough to avoid a "dragging" phenomenon similar to that sometimes experienced by "lazy" evaluation strategies (see [Pey87]).

3.1.2 Messages

A **message** is a vertex that has `target` and `tail` terminals, and the methods necessary to allow it to climb through `Copy` trees in the fashion described in section 2.4.2. Messages are used to model both procedure calling and message passing. Most messages also have a `cont` terminal that is connected to a compiler-produced continuation. Such messages are called **operations**.

The following message types are built in to the Scheme run-time system:

Type	Labels
`Return 1`	"`target`", "`tail`", "`0`"
`Call 0`	"`target`", "`tail`", "`cont`"
`Call 1`	"`target`", "`tail`", "`cont`", "`0`"
`Call 2`	"`target`", "`tail`", "`cont`", "`0`", "`1`"
`Call 3`	"`target`", "`tail`", "`cont`", "`0`", "`1`", "`2`"
`Call 4`	"`target`", "`tail`", "`cont`", "`0`", "`1`", "`2`", "`3`"
`Call 5`	"`target`", "`tail`", "`cont`", "`0`", "`1`", "`2`", "`3`", "`4`"
`Car`	"`target`", "`tail`", "`cont`"
`Cdr`	"`target`", "`tail`", "`cont`"
`Set Car`	"`target`", "`tail`", "`cont`", "`new`"
`Set Cdr`	"`target`", "`tail`", "`cont`", "`new`"
`Null?`	"`target`", "`tail`", "`cont`"

The one message that lacks a `cont` terminal (and is therefore not an operation) is `Return 1`. The reason for this is that `Return 1` is used to resume a continuation—typically one that was recently attached to the `cont` terminal of an operation. The `0` terminal of a `Return 1` is attached to the value that is to be returned to the continuation.[1]

[1]The "`1`" in the name "`Return 1`" reflects the fact that the run-time supports multiple return values. There also exist messages `Return 0`, `Return 2`, `Return 3`, etc., for returning

3.1.2.1 Operations

The operations `Call 0` through `Call 5` are used to invoke objects that represent procedures.[2] The arguments to be passed to the procedure are attached to the terminals `0` through `4`, and the continuation is attached to the `cont` terminal.

The `Car` operation is generated by calls to the Scheme `CAR` procedure. (How this happens is covered below in section 3.1.3.1.) The method

```
(graph (0 1 2 3)
  (<Car> target:4 tail:2 cont:3)
  (<Cons> 4 car:0 cdr:1))
(graph (0 1 2 3)
  (<Cons> 2 car:4 cdr:1)
  (<Return 1> target:3 tail:5 0:6)
  (<Drop> 5)
  (<Copy> target:0 a:6 b:4))
```

makes `Car` behave as it should when it climbs up to a `Cons`: The `car` of the `Cons` is copied, one copy is returned in a `Return 1` message sent to the continuation, and the other copy becomes the `car` of the recreated `Cons` that will be seen by the next message to arrive. An analogous method is defined for the `Cdr` operation.

The `Set Car` operation is generated by calls to the Scheme `SET-CAR!` procedure. The method

```
(graph (0 1 2 3 4)
  (<Set Car> target:5 tail:2 cont:3 new:4)
  (<Cons> 5 car:0 cdr:1))
(graph (0 1 2 3 4)
  (<Cons> 2 car:4 cdr:1)
  (<Return 1> target:3 tail:5 0:0)
  (<Drop> 5))
```

creates a new `Cons` whose `car` is the value that was previously attached to the `new` terminal of the `Set Car`. An analogous method is defined for the `Set Cdr` operation.

The `Null?` operation is generated by calls to the Scheme `NULL?` procedure. The methods

---

other numbers of values. The current compiler does not make use of these other messages, so they are not described here.

[2] Actually, the operation `Call` n is defined for all non-negative n, but none of the examples here will require more than 5 arguments.

```
(graph (0 1 2 3)
  (<Null?> target:4 tail:2 cont:3)
  (<Cons> 4 car:0 cdr:1))
(graph (0 1 2 3)
  (<Cons> 2 car:0 cdr:1)
  (<False> 4)
  (<Return 1> target:3 tail:5 0:4)
  (<Drop> 5))
```

and

```
(graph (0 1)
  (<Null?> target:2 tail:0 cont:1)
  (<Nil> 2))
(graph (0 1)
  (<Nil> 0)
  (<True> 2)
  (<Return 1> target:1 tail:3 0:2)
  (<Drop> 3))
```

allow the `Null?` operation to distinguish between `Cons` vertices and `Nil` vertices: A `True` or `False` vertex is returned to the continuation, as appropriate, and the original target is recreated for the benefit of the next message to arrive.

3.1.3 Atoms

An **atom** is a vertex that has a single terminal (its handle) and a method that makes a duplicate of the atom when it is connected to the `target` terminal of a `Copy` vertex. For example, the atomic type `True` has the associated method:

```
(graph (0 1)
  (<True> 2)
  (<Copy> target:2 a:0 b:1))
(graph (0 1)
  (<True> 1)
  (<True> 0))
```

You might expect that a method like

```
(graph ()
  (<True> 0)
  (<Drop> 0))
(graph ())
```

would be needed to make atoms disappear when they were dropped. However, as discussed in section 2.4, disconnected subgraphs can always be discarded without any effect on the output of the computation; instead of defining such methods, we can rely on disconnected subgraph garbage collection to clean up when atoms are dropped.

The following miscellaneous atomic types are built in to the Scheme run-time system:

Type	Labels
`Drop`	“”
`Nil`	“”
`True`	“”
`False`	“”
`Number` x	“”
`Failure`	“”
`Sink`	“”

`Drop` vertices were described in section 2.4. `Nil` vertices represent the empty list. `True` and `False` vertices are used as boolean values. For any number x there is an atomic vertex type named “`Number` x” that represents that value.

`Failure` and `Sink` are part of the support for `Future`s. A `Failure` is a `Future` that can never become a “real” value, and a `Sink` is a continuation that simply discards any value that is returned to it. The implementation of `Future`s is described more detail below.

3.1.3.1 Globals

Identifiers that occur free in top level Scheme expressions are called global identifiers. Global identifiers typically name primitive procedures such as `+`, `CONS`, `CAR` or `CALL-WITH-CURRENT-CONTINUATION` that have no internal state. In those cases we can make the values of these global identifiers be atoms that respond to the appropriate `Call` operation. Atoms that are the value of global identifiers are simply called **globals**. For any global identifier G there is a global vertex type named “`Global` G” that is the type of its value.

Constructors. Many global types respond to the appropriate `Call` operation by returning a new vertex of some other type. Such globals are called `constructors`. The scheme run-time supports the following constructors:

Type	Constructed type
`Global CONS`	`Cons`
`Global +`	`Sum`
`Global -`	`Difference`
`Global *`	`Product`
`Global =`	`Equal?`
`Global <`	`Less?`
`Global >`	`Greater?`

(The constructed types are all objects introduced in section 3.1.1.)

The method that controls what happens when a `Global CONS` vertex encounters a `Call 2` operation is typical of the behavior of constructors:

```
(graph (0 1 2 3)
  (<Global CONS> 4)
  (<Call 2> target:4 tail:0 cont:1 0:2 1:3))
(graph (0 1 2 3)
  (<Global CONS> 0)
  (<Cons> 4 car:2 cdr:3)
  (<Return 1> target:1 tail:5 0:4)
  (<Drop> 5))
```

This method constructs a new `Cons` vertex and returns it to the continuation, initializing its `car` and `cdr` from the arguments to the `Call 2`. It also leaves behind a `Global CONS` vertex in case another message is waiting on the `tail` of the `Call 2`. (This is necessary because the caller can't know whether the procedure being called is an atom or an object, so it must use the same calling convention in either case.)

Operators. Many global types respond to the appropriate `Call` operation by sending another operation to the first argument. Such globals are called `operators`. The scheme run-time supports the following operators:

Type	Operation sent
`Global CAR`	`Car`
`Global CDR`	`Cdr`
`Global SET-CAR!`	`Set Car`
`Global SET-CDR!`	`Set Cdr`
`Global NULL?`	`Null?`

The method that controls what happens when a `Global SET-CAR!` vertex encounters a `Call 2` operation is typical of the behavior of operators:

```
(graph (0 1 2 3)
  (<Global SET-CAR!> 4)
  (<Call 2> target:4 tail:0 cont:1 0:2 1:3))
(graph (0 1 2 3)
  (<Global SET-CAR!> 0)
  (<Set Car> target:2 tail:4 cont:1 new:3)
  (<Drop> 4))
```

This method creates a `Set Car` operation and targets it for the `Call`'s first argument. The `Call`'s second argument is attached to the `Set Car`'s `new` terminal, and the `Call`'s continuation becomes the continuation for the `Set Car`. As with the constructors, the `Global SET-CAR!` operator is reconstructed in case another message is waiting on the `tail` of the `Call 2`.

Call with current continuation. Many people find the procedure `CALL-WITH-CURRENT-CONTINUATION` to be scary. Those people may safely skip the rest of this section. On the other hand, I think that this implementation of `CALL-WITH-CURRENT-CONTINUATION` is particularly nice, so I encourage you to read and understand it. Perhaps afterwards you won't be scared any more!

The method

```
(graph (0 1 2)
  (<Global CALL-WITH-CURRENT-CONTINUATION> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global CALL-WITH-CURRENT-CONTINUATION> 0)
  (<Copy> target:1 a:4 b:6)
  (<Continuation> 3 cont:4)
  (<Call 1> target:2 tail:5 cont:6 0:3)
  (<Drop> 5))
```

serves to support Scheme's `CALL-WITH-CURRENT-CONTINUATION` procedure. Recall that the first argument to `CALL-WITH-CURRENT-CONTINUATION` is a one-argument procedure, which is to be tail-recursively applied to an escape procedure that captures the continuation that was originally passed to `CALL-WITH-CURRENT-CONTINUATION`. This method makes a vertex of type `Continuation` that holds one copy of the continuation in its `cont` terminal. The other copy is supplied as the continuation for the tail-recursive call of the one-argument procedure that was passed to `CALL-WITH-CURRENT-CONTINUATION`.

A `Continuation` is an object type (see section 3.1.1). Since a `Continuation` must behave as a one-argument procedure, the following method describes the interaction of a `Continuation` with a `Call 1`:

```
(graph (0 1 2 3)
  (<Continuation> 4 cont:0)
  (<Call 1> target:4 tail:1 cont:2 0:3))
(graph (0 1 2 3)
  (<Drop> 2)
  (<Continuation> 1 cont:4)
  (<Return 1> target:0 tail:4 0:3))
```

This method takes the argument passed in the `Call`, and returns it directly to the captured continuation. As usual, the `Continuation` vertex is reconstructed for the benefit of any future callers.

There are two interesting things to notice about this method. First, observe that when a `Continuation` is called, the continuation supplied by the `Call` operation is dropped. This method, and the one immediately preceding it that copied a continuation, are the *only* methods that treat continuations nonlinearly.

Second, notice that when the `Continuation` is reconstructed, no `Copy` vertex is used to duplicate the captured continuation. Instead, the `tail` of the

`Return 1` vertex is used. This works because the captured continuation also obeys the protocol for an object, so after it has processed the `Return 1` it will connect a reconstruction of itself to the `tail` terminal. This device will appear many times in subsequent methods—in effect, every message contains an implicit `Copy` that the sender of a message can use if it wishes to retain its access to the target.

3.1.4 Futures

One type fails to fit into the neat categorization of types into objects, messages, and atoms: the `Future` vertex type. A `Future` is created by the (extended) Scheme `FUTURE` special form. A `Future` has two terminals, labeled "" and "`as cont`", *both* of which can be thought of as handles. The "" terminal is connected to the parts of the working graph that are treating the future as if it were already a full-fledged value. For example, this terminal might be connect to the `left` terminal of a `Sum`, or the `car` terminal of a `Cons`. The "`as cont`" terminal is connected to the parts of the working graph that are working on computing the future's eventual value. This terminal functions as a continuation, so it accepts `Return 1` messages through the following method:

```
(graph (0 1 2)
  (<Future> 2 as cont:3)
  (<Return 1> target:3 tail:0 0:1))
(graph (0 1 1)
  (<Sink> 0))
```

This method connects the returned value directly to the parts of the graph that have proceeded on as if the value was already there. In effect the `Future` vertex "becomes" the returned value. This method also replaces the `Future` with a `Sink` atom. A `Sink` responds to any additional attempts to return values by simply dropping them:[3]

```
(graph (0 1)
  (<Sink> 2)
  (<Return 1> target:2 tail:0 0:1))
(graph (0 1)
  (<Sink> 0)
  (<Drop> 1))
```

Of course this can only occur if `CALL-WITH-CURRENT-CONTINUATION` was used to capture and duplicate the continuation—normally the `Sink` atom will be quickly dropped.

If *no* values are ever returned to the "`as cont`" terminal of a `Future`, i.e. if it is simply dropped, then this method will apply:

[3]As a result, the futures described here behave like those in MultiLisp [Hal84]. Other possible behaviors are described in [Mil87, KW90].

```
(graph (0)
  (<Future> 0 as cont:1)
  (<Drop> 1))
(graph (0)
  (<Failure> 0))
```

So the `Future` vertex becomes a `Failure` atom. There are no methods that allow a Scheme program to test for such a failed future, this is done purely as a storage reclamation measure.

For similar reasons, the method

```
(graph (0)
  (<Future> 1 as cont:0)
  (<Drop> 1))
(graph (0)
  (<Sink> 0))
```

handles the case where the "" terminal of a `Future` is dropped.

3.2 Translating Scheme constructs

This section describes how a Scheme program is translated into a collection of methods.

The only top-level expressions that are supported are procedure definitions. Each definition defines a global type (see section 3.1.3.1) and a method for what should happen when the global type is treated as a procedure. For example, given the top-level definition

```
(define (inc x) (+ x 1))
```

the method

```
(graph (0 1 2)
  (<Global INC> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global INC> 0)
  (<Number 1> 3)
  (<Sum> 4 left:2 right:3)
  (<Return 1> target:1 tail:5 0:4)
  (<Drop> 5))
```

is the result. It is easy to see that this method performs the computation called for in the body of the definition and returns the answer to the supplied continuation. (Notice that the compiler has integrated knowledge about the meaning of the global identifier `+` into this method.)

Definitions that call for more complicated computations will require the compiler to generate new object types to represent the required closures and continuations. For example, compiling

```
(define (f x) (lambda (y) (+ x y)))
```

yields the method

```
(graph (0 1 2)
  (<Global F> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global F> 0)
  (<Lambda 1583> 3 x:2)
  (<Return 1> target:1 tail:4 0:3)
  (<Drop> 4))
```

which creates a `Lambda 1583`[4] object to serve as a closure. The value of `X` is captured in this closure for later use when the `Lambda 1583` handles a `Call 1` operation:

```
(graph (0 1 2 3)
  (<Lambda 1583> 4 x:3)
  (<Call 1> target:4 tail:0 cont:1 0:2))
(graph (0 1 2 3)
  (<Lambda 1583> 0 x:4)
  (<Copy> target:3 a:6 b:4)
  (<Sum> 5 left:6 right:2)
  (<Return 1> target:1 tail:7 0:5)
  (<Drop> 7))
```

This adds the argument to one copy of the captured value, and recreates the `Lambda 1583` using the other copy.

Note that the values of the captured variables are held directly by the closure object itself. There are no separate environment objects as there are in a traditional Scheme interpreter [AS85]. Environments are not needed when there are no local variable assignments. (Recall that we are pretending that assignment statements have been eliminated through assignment conversion.)

Continuations are created in a similar manner. For example, compiling

```
(define (f g) (+ (g) 1))
```

yields the method

[4] This is a badly chosen name. It would be better if the compiler gave such generated types names that started with "`Procedure`" or "`Closure`" instead of "`Lambda`".

```
(graph (0 1 2)
  (<Global F> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global F> 0)
  (<Evarg 1472> 3 cont:1)
  (<Call 0> target:2 tail:4 cont:3)
  (<Drop> 4))
```

which creates an `Evarg 1472` object to serve as the continuation when the procedure `G` is called. An `Evarg 1472` handles a `Return 1` message by incrementing the returned value before passing that value back to the original continuation:

```
(graph (0 1 2)
  (<Evarg 1472> 3 cont:2)
  (<Return 1> target:3 tail:0 0:1))
(graph (0 1 2)
  (<Evarg 1472> 0 cont:3)
  (<Return 1> target:2 tail:3 0:5)
  (<Sum> 5 left:1 right:4)
  (<Number 1> 4))
```

Like any object, an `Evarg 1472` reconstructs itself for the benefit of any further `Return 1` messages.

Note that in order to properly reconstruct itself, this continuation method needs to make a copy of the original continuation it holds. (It does this by using the `tail` of the `Return 1`.) This is in apparent violation of my assertion in section 3.1.3.1 that only the methods associated with `CALL-WITH-CURRENT-CONTINUATION` ever treat continuations in a nonlinear fashion.

However, observe that this copying is only done to cover the case where the `Evarg 1472` is *itself* copied. In the absence of `CALL-WITH-CURRENT-CONTINUATION` every `Return 1` message has a `Drop` following close behind on its `tail`, and these continuations are all quickly discarded. If `CALL-WITH-CURRENT-CONTINUATION` were eliminated from the language, continuation methods would not need to engage in all this useless duplication.

Continuations can also capture variables from the lexical environment and intermediate values. Consider the definition:

```
(define (bar f) (+ (f) (f)))
```

This compiles into the method

```
(graph (0 1 2)
  (<Global BAR> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global BAR> 0)
  (<Evarg 673> 3 cont:1 f:4)
  (<Call 0> target:2 tail:4 cont:3))
```

which calls the function `F` with a continuation of type `Evarg 673` which holds both the original continuation, and a second copy of the value of `F`. When a value is returned to an `Evarg 673`, the method

```
(graph (0 1 2 3)
  (<Evarg 673> 4 cont:2 f:3)
  (<Return 1> target:4 tail:0 0:1))
(graph (0 1 2 3)
  (<Evarg 673> 0 cont:4 f:5)
  (<Evarg 662> 6 cont:7 0:1)
  (<Call 0> target:3 tail:5 cont:6)
  (<Copy> target:2 a:7 b:4))
```

calls the saved value of `F` (for the second time) with a continuation of type `Evarg 662` which maintains a hold on the original continuation, and also holds on to the returned value. When a value is returned to an `Evarg 662`, the method

```
(graph (0 1 2 3)
  (<Evarg 662> 4 cont:2 0:3)
  (<Return 1> target:4 tail:0 0:1))
(graph (0 1 2 3)
  (<Evarg 662> 0 cont:4 0:5)
  (<Sum> 6 left:7 right:1)
  (<Return 1> target:2 tail:4 0:6)
  (<Copy> target:3 a:7 b:5))
```

adds the returned value value to the saved value and returns the result to the saved continuation.

The examples presented so far demonstrate how the basic Scheme constructs can be compiled into methods. Combinations, local and global identifiers, `LAMBDA` expressions, and constants have all been demonstrated. The following subsections describe the treatment of the Scheme special forms `IF`, `BEGIN`, `LETREC` and `FUTURE`. (The special forms `LET`, `LET*`, `COND`, `AND` and `OR` are also present. They are defined as macros.) This should complete the picture of how an arbitrary sequential Scheme program can be compiled into a linear graph grammar.

3.2.1 BEGIN

`BEGIN` is handled in much the same way as a combination. For example,

```
(define (twice f) (begin (f) (f)))
```

generates the method

```
(graph (0 1 2)
  (<Global TWICE> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global TWICE> 0)
  (<Evseq 733> 3 cont:1 f:4)
  (<Call 0> target:2 tail:4 cont:3))
```

which calls the function `F` with a continuation of type `Evseq 733` which holds both the original continuation, and a second copy of the value of `F`. When a value is returned to an `Evseq 733`, the method

```
(graph (0 1 2 3)
  (<Evseq 733> 4 cont:2 f:3)
  (<Return 1> target:4 tail:0 0:1))
(graph (0 1 2 3)
  (<Evseq 733> 0 cont:4 f:5)
  (<Call 0> target:3 tail:5 cont:6)
  (<Drop> 1)
  (<Copy> target:2 a:6 b:4))
```

drops the returned value and calls the saved value of `F` (for the second time) with the original continuation.

3.2.2 IF

Conditional expressions are somewhat more complicated. There are two cases, depending on the nature of the expression which is to be tested. In some cases the compiler can produce a single type and two methods, while in other cases the compiler is forced to generate an additional type and an additional method.

First, an example of the simple case: The definition

```
(define (test n x) (if (< n 2) x 105))
```

generates the method

```
(graph (0 1 2 3)
  (<Global TEST> 4)
  (<Call 2> target:4 tail:0 cont:1 0:2 1:3))
(graph (0 1 2 3)
  (<Global TEST> 0)
  (<Test 1307> 4 cont:1 x:3)
  (<Less?> 4 left:2 right:5)
  (<Number 2> 5))
```

which compares `N` with 2, and connects the result to the handle of a `Test 1307` vertex. `Test 1307` is a compiler generated type, similar to a continuation. It captures the continuation and the values of the variables that are used on either arm of the conditional. A `Test 1307` is *not* a continuation, because it does not expect to handle a `Return 1` message. Instead it expects to be connected to the value to be tested, so that it may perform a dispatch on the Boolean value. If the value is `True`, the method

```
(graph (0 1)
  (<Test 1307> 2 cont:0 x:1)
  (<True> 2))
(graph (0 1)
  (<Return 1> target:0 tail:2 0:1)
  (<Drop> 2))
```

returns the preserved value of `X` to the continuation. If the value is `False`, the method

```
(graph (0 1)
  (<Test 1307> 2 cont:0 x:1)
  (<False> 2))
(graph (0 1)
  (<Drop> 1)
  (<Number 105> 2)
  (<Return 1> target:0 tail:3 0:2)
  (<Drop> 3))
```

discards the preserved value of `X` and returns 105.

The previous example was simple because the compiler was able to avoid generating an explicit continuation for the call to `<`. This won't always be possible. If the example had been

```
(define (test f x) (if (f) x 105))
```

then the compiler would have generated the method

```
(graph (0 1 2 3)
  (<Global TEST> 4)
  (<Call 2> target:4 tail:0 cont:1 0:2 1:3))
(graph (0 1 2 3)
  (<Global TEST> 0)
  (<Evif 1450> 4 cont:1 x:3)
  (<Call 0> target:2 tail:5 cont:4)
  (<Drop> 5))
```

which calls `F` with a continuation of type `Evif 1450`. When a value is returned to an `Evif 1450` the method

```
(graph (0 1 2 3)
  (<Evif 1450> 4 cont:2 x:3)
  (<Return 1> target:4 tail:0 0:1))
(graph (0 1 2 3)
  (<Evif 1450> 0 cont:4 x:5)
  (<Test 1307> 1 cont:6 x:7)
  (<Copy> target:2 a:6 b:4)
  (<Copy> target:3 a:7 b:5))
```

connects that value to the `Test 1307` vertex and (as usual) recreates the continuation.

3.2.3 FUTURE

Since the `FUTURE` special form is not a standard part of Scheme, we present a brief introduction of it here. (More complete descriptions can be found elsewhere [Hal84, Mil87].) A `FUTURE` expression contains a single subexpression. When the `FUTURE` expression is evaluated, what was previously a single thread of execution splits, and execution continues concurrently in two different directions. One thread of execution starts evaluating the subexpression. The other thread of execution continues the execution of the rest of the program, taking as the value of the `FUTURE` expression something called a "future". Execution continues in parallel until one of two things happens: If the evaluation of the subexpression yields a value, then the future *becomes* that value.[5] If, on the other hand, the evaluation of the rest of the program requires an actual value instead of a future, then that thread simply waits until evaluation of the subexpression finally yields a value.

The compilation of the `FUTURE` special form is quite simple. Consider

```
(define (future-call f) (future (f)))
```

which compiles into the single method:

[5] An operation that can prove quite challenging to implement!

```
(graph (0 1 2)
  (<Global FUTURE-CALL> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global FUTURE-CALL> 0)
  (<Future> 3 as cont:4)
  (<Return 1> target:1 tail:5 0:3)
  (<Call 0> target:2 tail:6 cont:4)
  (<Drop> 5)
  (<Drop> 6))
```

Almost exactly the same method would have been generated if the call to `F` had not been wrapped in a `FUTURE`. The difference is the additional `Future` and `Return 1` vertices strung together between the original continuation and the `cont` terminal of the `Call 0`. This has the effect of immediately returning the `Future` to the caller, while allowing the call to `F` to proceed simultaneously.

Various methods for `Future`s were described in section 3.1.4.

You might think of `Future` vertices as anti-particles for `Return 1` vertices. When a `FUTURE` special form is evaluated, a `Return 1` and an anti-`Return 1` (a `Future`) are created. Later, when the expression inside the `FUTURE` produces a `Return 1`, the `Return 1` and the anti-`Return 1` annihilate each other.

3.2.4 LETREC

There are many interesting things to say about the implementation of `LETREC`, but none of them is really essential to an understanding of the current system, so the reader can skip this subsection without missing anything important. On the other hand, Scheme language aficionados will find this stuff right up their alley.

Let us start with a simple example: The definition

```
(define (make-f)
  (letrec ((f (lambda (n) (f (+ n 1)))))
    f))
```

generates the method

```
(graph (0 1)
  (<Global MAKE-F> 2)
  (<Call 0> target:2 tail:0 cont:1))
(graph (0 1)
  (<Global MAKE-F> 0)
  (<Lambda 2101> 2 f:3)
  (<Return 1> target:1 tail:4 0:5)
  (<Drop> 4)
  (<Copy> target:2 a:3 b:5))
```

which creates a closure of type `Lambda 2101`. One copy of the closure is returned and the other copy is looped back to become the value of the closed-over variable `F`. Later, when the closure is called, the method

```
(graph (0 1 2 3)
  (<Lambda 2101> 4 f:3)
  (<Call 1> target:4 tail:0 cont:1 0:2))
(graph (0 1 2 3)
  (<Lambda 2101> 0 f:4)
  (<Number 1> 5)
  (<Sum> 6 left:2 right:5)
  (<Call 1> target:3 tail:4 cont:1 0:6))
```

calls one copy of the function `F`, and makes a second copy for the reconstructed `Lambda 2101` vertex.

So far, this is exactly what one would expect given the usual Scheme definition for `LETREC` and the way Scheme objects are being modeled, but the underlying mechanism that supports this is actually much more general. For example

```
(define (circular-list x)
  (letrec ((l (cons x l)))
   l))
```

generates the method

```
(graph (0 1 2)
  (<Global CIRCULAR-LIST> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global CIRCULAR-LIST> 0)
  (<Cons> 3 car:2 cdr:4)
  (<Return 1> target:1 tail:5 0:6)
  (<Drop> 5)
  (<Copy> target:3 a:4 b:6))
```

which returns (one copy of) a `Cons` whose `cdr` is (the other copy of) itself.

In fact, by using `Futures`, *any* expression can appear in a `LETREC`. For example

```
(define (fixpoint f)
  (letrec ((x (f x)))
   x))
```

generates the method

```
(graph (0 1 2)
  (<Global FIXPOINT> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global FIXPOINT> 0)
  (<Future> 3 as cont:4)
  (<Return 1> target:1 tail:5 0:6)
  (<Call 1> target:2 tail:7 cont:4 0:8)
  (<Drop> 5)
  (<Drop> 7)
  (<Copy> target:3 a:8 b:6))
```

which creates a `Future` to fill in for the value of the call to `F`, and then passes one copy in to `F` as its argument, and returns the other copy.

The first two examples were produced by generating the fully general translation, using `Future`s, and then optimizing the `Future`s away (using the techniques described in the next section). In the usual case, where the expressions in a `LETREC` are all `LAMBDA`-expressions, the futures can always be eliminated.

The fact that futures can be used to implement a fully general `LETREC` is not new. See [Mil87].

3.3 Optimization: simulation

The compiler uses one simple technique to optimize linear graph grammars: It applies methods to the right hand sides of other methods.

To see why this is a safe thing to do, suppose method B applies to the right hand side of method A. In other words, the left hand side of B occurs as a subgraph of the right hand side of A. When A is actually applied to the working graph at run-time, its right hand side will be instantiated as a subgraph of the working graph; so B's left hand side will now occur in the working graph. When A is applied, B *always* becomes applicable immediately afterwards. Applying B to A simply performs that application at compile-time. In effect, this does a compile-time **simulation** of the run-time world.

Some possible execution histories are eliminated by making these decisions at compile-time. After A is applied, B becomes applicable, but the run-time is free to choose some other applicable method C instead. C might even change the graph so that B is no longer applicable. By making the choice to apply B at compile-time, that possibility is precluded. Fortunately, the compiler takes care to generate grammars that work correctly given any scheduling of methods, so this isn't a problem.

Simulation turns out to be quite good at cleaning up the rubbish left behind by the raw source-to-methods translation algorithm. In fact, all the methods presented in the previous section as examples of compiler output were already optimized in this way. Unsimulated methods are considerably more difficult to read. For example, the definition

```
(define (invoke f) (f))
```

initially becomes two methods:

```
(graph (0 1 2)
  (<Global INVOKE> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global INVOKE> 0)
  (<Evarg 317> 3 cont:1)
  (<Return 1> target:3 tail:4 0:2)
  (<Drop> 4))

(graph (0 1 2)
  (<Evarg 317> 3 cont:2)
  (<Return 1> target:3 tail:0 0:1))
(graph (0 1 2)
  (<Evarg 317> 0 cont:3)
  (<Call 0> target:1 tail:4 cont:5)
  (<Drop> 4)
  (<Copy> target:2 a:5 b:3))
```

The object type `Evarg 317` is the continuation for the evaluation of the expression `F`. The first method creates an `Evarg 317` and immediately returns the value of `F` to it. The second method describes what happens when a value is returned to an `Evarg 317`. So the second method can be applied to the right hand side of the first method so that it becomes:

```
(graph (0 1 2)
  (<Global INVOKE> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global INVOKE> 0)
  (<Copy> target:1 a:5 b:6)
  (<Evarg 317> 4 cont:6)
  (<Call 0> target:2 tail:7 cont:5)
  (<Drop> 4)
  (<Drop> 7))
```

Then we can apply the method that applies when a `Drop` is connected to the handle of a `Evarg 317`. (Recall from section 3.1.1 that all object types have such a method.) The result:

```
(graph (0 1 2)
  (<Global INVOKE> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global INVOKE> 0)
  (<Copy> target:1 a:5 b:6)
  (<Call 0> target:2 tail:7 cont:5)
  (<Drop> 6)
  (<Drop> 7))
```

Now we can apply the method that applies when a `Drop` is connected to the `b` terminal of a `Copy` (see section 2.4) to obtain the final result:

```
(graph (0 1 2)
  (<Global INVOKE> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global INVOKE> 0)
  (<Call 0> target:2 tail:7 cont:1)
  (<Drop> 7))
```

Simulation sometimes looks a lot like β-reduction, but it is both more and less powerful. It is *more* powerful because it is able to integrate knowledge of the behavior of various data types into the generated code. For example, the definition

```
(define (f x) (car (cons x x)))
```

becomes simply

```
(graph (0 1 2)
  (<Global F> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global F> 0)
  (<Return 1> target:1 tail:3 0:2)
  (<Drop> 3))
```

because the compiler has the complete set of methods describing the behavior of `CAR` and `CONS` available at compile-time. (This example also illustrates how nonlinearities can sometimes be eliminated at compile-time.)

On the other hand, simulation is *less* powerful than β-reduction because it is unable to eliminate captured variables from closures. For example

```
(define (7up) ((lambda (x) (lambda () x)) 7))
```

can be β-substituted to become

```
(define (7up) (lambda () 7))
```

but compiles into the following two methods:

```
(graph (0 1)
  (<Global 7UP> 2)
  (<Call 0> target:2 tail:0 cont:1))
(graph (0 1)
  (<Global 7UP> 0)
  (<Lambda 487> 2 x:3)
  (<Number 7> 3)
  (<Return 1> target:1 tail:4 0:2)
  (<Drop> 4))

(graph (0 1 2)
  (<Lambda 487> 3 x:2)
  (<Call 0> target:3 tail:0 cont:1))
(graph (0 1 2)
  (<Lambda 487> 0 x:3)
  (<Return 1> target:1 tail:4 0:5)
  (<Drop> 4)
  (<Copy> target:2 a:5 b:3))
```

We would prefer for the compiler to eliminate the variable `X` and generate instead the two methods:

```
(graph (0 1)
  (<Global 7UP> 2)
  (<Call 0> target:2 tail:0 cont:1))
(graph (0 1)
  (<Global 7UP> 0)
  (<Lambda 487> 2)
  (<Return 1> target:1 tail:3 0:2)
  (<Drop> 3))

(graph (0 1)
  (<Lambda 487> 2)
  (<Call 0> target:2 tail:0 cont:1))
(graph (0 1)
  (<Lambda 487> 0)
  (<Return 1> target:1 tail:2 0:3)
  (<Drop> 2)
  (<Number 7> 3))
```

The problem is that the initial translation decided that the type `Lambda 487` (used to represent closures of the expression `(LAMBDA () X)`) needed an `x` terminal to remember the value of `X`, and no amount of mere simulation can eliminate a terminal. More complex optimizations would be required to duplicate full β-reduction.

Another important function served by simulation is to integrate calls to the constructors and operators described in section 3.1.3.1 into the code. Calls to `+`, for example, are converted into `Sum` vertex constructions, rather than remaining explicit calls to `Global +`. Many of the raw translations also rely on simulation to produce reasonable code. For example, the raw translation for a `LETREC` always makes use of futures, and assumes they will be eliminated in the common cases (section 3.2.4).

Simulation remains a valid technique even if the compiler uses only a subset of the methods and types that will be present at run-time. In fact, the current compiler doesn't have complete knowledge of the numeric types and the methods that implement arithmetic (although it could, given some more work).

The compiler also lacks all knowledge of the vertex types used at run-time to implement I/O. Such types cannot even occur in any of the methods manipulated at compile-time. Therefore, in particular, such types cannot occur in any disconnected component appearing in the right hand side of a method. Thus the compiler knows that *all* disconnected components in method right hand sides will be garbage at run-time (see section 2.4), and so it can discard them immediately.

Of course there is a danger in doing such a complete simulation of the program at compile-time: The compiler might find itself running the program to completion using the considerably slower compile-time data structures. (For example, if the expression `((LAMBDA (X) (X X)) (LAMBDA (X) (X X)))` appears in the program.) Compilers that do β-reduction face this danger as well. To prevent the simulation from getting out of hand, no right hand side is ever simulated for more than 250 steps. That number was picked because a couple of contrived examples really needed to run that long, but ordinarily all right hand sides terminate in less than 20 steps.

Other optimizations besides simulation are certainly possible. Simulation is particularly easy because it only involves the interaction of two methods (the method being applied, and the method whose right hand side is being modified).

3.4 A real example

So far, the examples have all been chosen to emphasize some particular aspect of compilation. To get a feeling for what compilation will do with a more typical Scheme procedure, consider the following famous function:

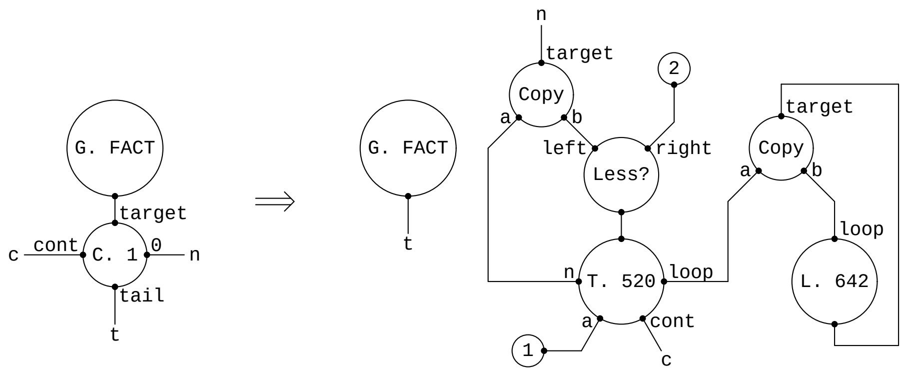

Figure 3.2: The method for calling **FACT**

```
(define (fact n)
  (let loop ((a 1) (n n))
    (if (< n 2)
        a
        (loop (* n a) (- n 1)))))
```

After translation and optimization, four methods remain. One method describes the behavior of when **FACT** is first called:

```
(graph (0 1 2)
  (<Global FACT> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global FACT> 0)
  (<Copy> target:2 a:7 b:10)
  (<Lambda 642> 3 loop:4)
  (<Copy> target:3 a:8 b:4)
  (<Test 520> 6 cont:1 n:7 loop:8 a:9)
  (<Number 1> 9)
  (<Less?> 6 left:10 right:5)
  (<Number 2> 5))
```

Among other things, this method builds a closure (a **Lambda 642** object) to represent the internal procedure named **LOOP** in the source. A picture of this method is shown in figure 3.2.[6] When this closure is invoked, the relevant method will be:

[6] The vertex types in this figure, and those that follow, have been abbreviated in a straightforward way. "**Lambda 642**" becomes "**L. 642**", "**Call 1**" becomes "**C. 1**", etc.

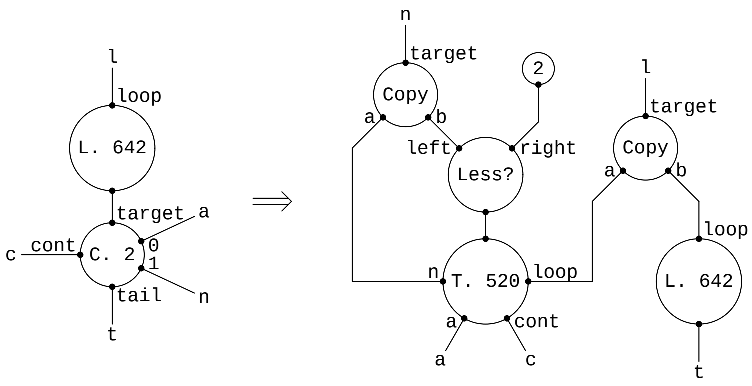

Figure 3.3: The method for calling `LOOP`

```
(graph (0 1 2 3 4)
  (<Lambda 642> 5 loop:4)
  (<Call 2> target:5 tail:0 cont:1 0:2 1:3))
(graph (0 1 2 3 4)
  (<Lambda 642> 0 loop:5)
  (<Number 2> 6)
  (<Test 520> 7 cont:1 n:8 loop:9 a:2)
  (<Less?> 7 left:10 right:6)
  (<Copy> target:3 a:8 b:10)
  (<Copy> target:4 a:9 b:5))
```

A picture of this method is shown in figure 3.3.

Notice that both of these methods contain code that checks to see if the current value of `N` is less than 2. The reason for this is that the aggressive simulation described in the previous section has the effect of open-coding calls to procedures known to the compiler—the first method of such a known procedure, the method that describes what happens immediately after the procedure is called, will be integrated into the method that called it.

These methods also both build `Test 520` vertices to react to the results of the comparison. The other two methods describe the two possible outcomes: Either `N` was less than 2:

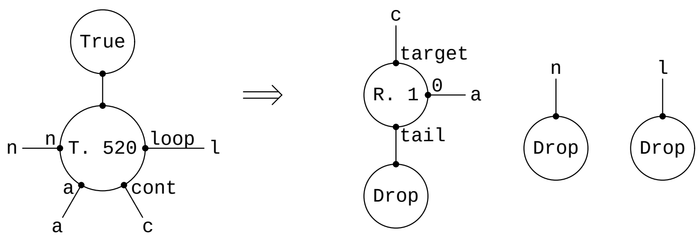

Figure 3.4: The method for when `(< N 2)` is true

```
(graph (0 1 2 3)
  (<Test 520> 4 cont:0 n:1 loop:2 a:3)
  (<True> 4))
(graph (0 1 2 3)
  (<Return 1> target:0 tail:4 0:3)
  (<Drop> 4)
  (<Drop> 1)
  (<Drop> 2))
```

in which case the current value of `A` is returned, or another trip around the loop is required:

```
(graph (0 1 2 3)
  (<Test 520> 4 cont:0 n:1 loop:2 a:3)
  (<False> 4))
(graph (0 1 2 3)
  (<Number 1> 4)
  (<Product> 5 left:6 right:3)
  (<Difference> 7 left:8 right:4)
  (<Call 2> target:2 tail:9 cont:0 0:5 1:7)
  (<Drop> 9)
  (<Copy> target:1 a:6 b:8))
```

in which case some arithmetic is performed to obtain new values to be arguments to `LOOP`. Pictures of these two methods are shown in figure 3.4 and figure 3.5. (Note that simulation was unable to discover that this call to `LOOP` is the same every time. This demonstrates how simulation is not quite good enough to remove all traces of `LETREC`. This failure is similar to the example on page 56, where the compiler failed to perform β-reduction.)

Figure 3.6 shows a piece of linear graph structure set up for a call to the `FACT` procedure. After applying the method for calling `FACT` (figure 3.2), and after copying the argument and performing the comparison the result appears in figure 3.7. The method for the false arm of the conditional (figure 3.5) can now

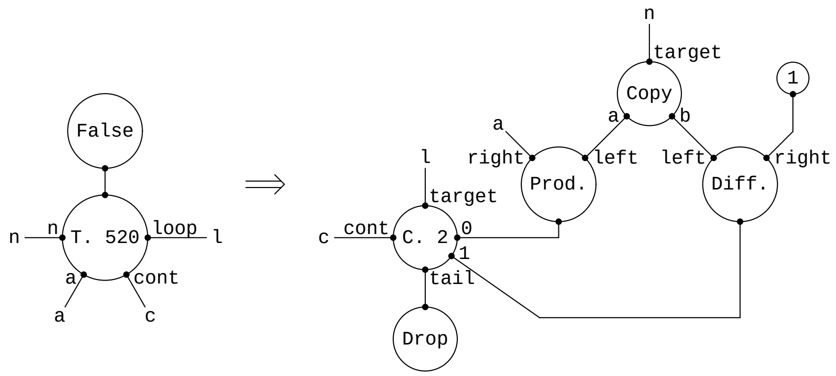

Figure 3.5: The method for when `(< N 2)` is false

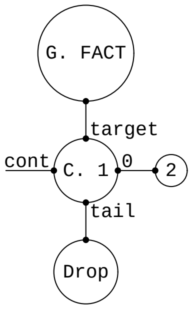

Figure 3.6: The initial working graph for `(FACT 2)`

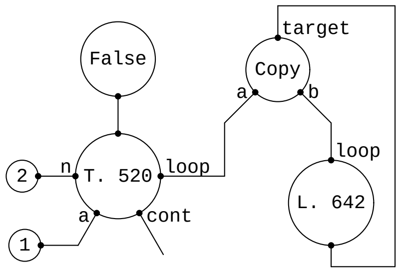

Figure 3.7: Before the conditional test (first time)

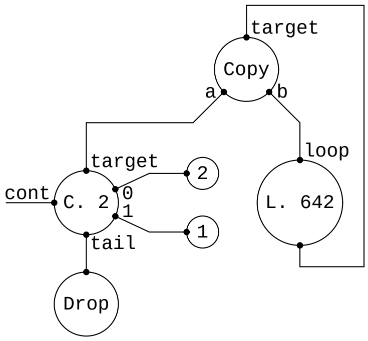

Figure 3.8: Before tree-climbing

be applied, and after some copying and some arithmetic the result appears in figure 3.8. Notice that the `Copy` and `Lambda 642` subgraph survived this round unchanged.

Now the `Call 2` vertex is connected to the `a` terminal of a `Copy`, so the standard tree-climbing method can be applied and this graph becomes the graph shown in figure 3.9. The `Drop` and `Copy` vertices will now be eliminated, and the method for calling a `Lambda 642` will run, resulting in another copy and comparison, all resulting in the graph shown in figure 3.10. Notice how the `Copy` and `Lambda 642` subgraph has been *recreated* by this process, and as a result this graph strongly resembles the one shown in figure 3.7.

This time of course, the comparison came out the other way, so the method for the true arm of the conditional now applies (figure 3.4), resulting in the graph shown in figure 3.11. The correct result is now being returned to the original continuation. Two disconnected components remain that will be garbage collected.

3.5 Code generation

Code generation has the following phases:

1. Compute which types and methods will be needed at run-time.

   Starting with the set of types known to be in the initial graph, the compiler looks for methods that might apply to a graph built from just those types, i.e. methods whose left hand side is made entirely from types in the set. Then the compiler adds all the types used in the right hand sides of these methods into the set. This operation is repeated until no new types and methods are discovered. (This "tree shake" will even discard built-in types and methods described in section 3.1 if they are not needed. If a

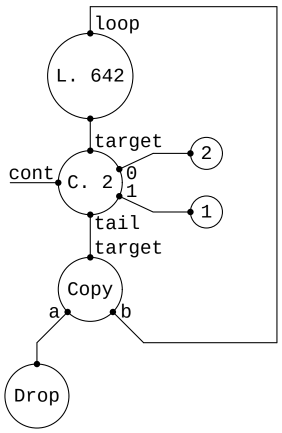

Figure 3.9: After tree-climbing

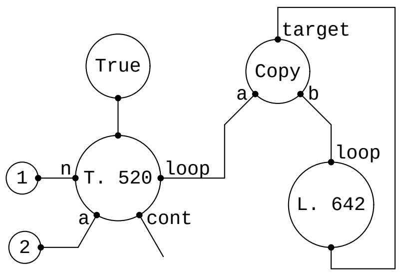

Figure 3.10: Before the conditional test (second time)

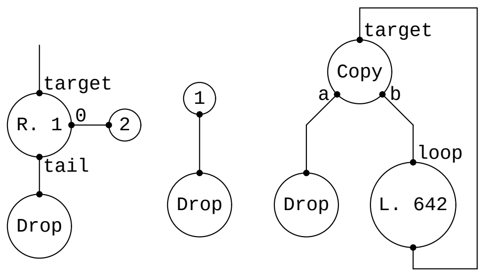

Figure 3.11: Returning the result

program doesn't use futures or `CONS`-cells, then none of the support for those features will be included.)

2. For each type discovered in the first phase, decide how vertices of that type will be represented at run-time.

   The run-time system uses tagged values to implement the connections between vertices. Connections to univalent types are represented as immediate tagged values, where the tag identifies the type of the vertex, and the rest of the value is ignored. For connections to larger valence types, the tag identifies the type of the vertex and which of its terminals is being connected to, and the rest of the value is the address of a block of storage that contains values that represent the vertex's other connections.

3. Write a C procedure that implements each method.

   Each method becomes a C procedure that performs surgery on the run-time working graph to transform a single instance of the method's left hand side into an instance of its right hand side. The C code calls various utilities provided by the run-time system to manipulate graph structure, and carefully alerts the run-time system whenever it creates a subgraph to which some other method might apply.

The set of types and methods computed in the first phase are used to compute the "activity" of each terminal of each type. A terminal is **active** if it occurs on one end of a connection in the left hand side of any method. An **inactive** terminal is one that is not active. A terminal is **monoactive** if it is the sole active terminal among all the terminals of its type. If an active terminal is not monoactive, then it is **polyactive**.

Recall that all methods are binary methods. The left hand side of a binary method is entirely characterized by the pair of terminals joined by its single connection. So for example, the method on page 38 describes what happens when the `target` terminal of a `Car` operation is joined to the handle of a `Cons` object. Those two terminals are thus active whenever this method is present at run-time. Furthermore, since no other methods or method schema will make any of the other terminals of the types `Car` and `Cons` active, those two terminals are in fact monoactive.

All three terminals of a `Copy` vertex are polyactive, because every message type has methods for when its handle is joined to either the `a` or the `b` terminal of a `Copy`, and in addition every atom type has a method for when it is joined to the `target` terminal of a `Copy`.

Terminal activity information plays a number of important roles:

- The compiler uses terminal activities when it designs the run-time representations used for vertices. More about this below.

- The linker uses terminal activities to check that the assumptions made in separately compiled modules about the characteristics of vertex types held in common are all consistent. Since the current compiler is actually a block compiler, this problem reduces to checking that the compiler agrees with the run-time system about the characteristics of a few built-in types. (Most of them, such as `Sum`, `Product` and the `Number` types, are concerned with implementing arithmetic.)

- The run-time system uses terminal activities to help make decisions about how to migrate graph structure around the network during distributed graph reduction. More about this in chapter 5.

The layout of the block of storage used to represent a vertex type can be varied depending on the requirements of the methods that mention that type in either their left or right hand side. The code generated for those methods is the *only* code that ever manipulates that storage, so the format of its contents is entirely up to the compiler. This leaves a fair amount of room for optimizations, but as this dissertation isn't concerned with the efficient *local* execution of linear graph grammars, these optimizations are not described here.

Instead I will assume that the compiler uses the general case representation for all types. In the general case a vertex of valence N is represented using an array of N tagged values that indicate for each terminal what other terminal it is connected to. Thus every connection is represented using a pair of pointers, one running in each direction.

The individual method procedures all share a fairly simple structure:

1. Allocate the storage needed to represent the vertices in the right hand side of the method.

2. Move connections between the other terminals of the old pair of vertices and the appropriate terminals of the newly allocated vertices.

3. Create additional internal connections between the newly allocated vertices.

4. Free the storage that was used to represent the two left hand side vertices.

In steps 2 and 3 connections to old terminals are broken and connections to new terminals are made. If the terminal involved is active, this can cause previously applicable methods to become inapplicable and vice versa. In these cases the method procedure makes or breaks the connection by calling an appropriate run-time routine that does the work necessary to maintain a queue of redexes. (This redex queue will be a major character in chapter 5.)

Methods could be compiled into considerably better code. It should be possible to produce methods that perform type dispatches and arithmetic directly, rather than always relying on the run-time. I see no fundamental obstacles that prevent a sufficiently good compiler from producing essentially the same code as any other Scheme compiler for the same source code. Achieving that goal remains a topic for future research.

The resulting collection of C procedures are compiled by the C compiler[7] and linked together with the run-time system. When the resulting executable file is started it initializes the storage system, starts up the network interface routines, loads all the method procedures into a hash table, builds the initial working graph (based on parsing the command line arguments), and calls the main scheduler loop (described in the chapter 5).

3.6 Summary

In this chapter I demonstrated how Scheme can be modeled using linear graph reduction. I showed how the various Scheme data types could be represented using linear graph structure, and how the Scheme language could be faithfully translated into a linear graph grammar. The current system is far from being a complete Scheme system—it lacks many data types and language features—but I tried to include enough of the picture so that it should be obvious how to finish the job.

A key property of this translation is that linear constructs in the original program are translated directly into linearities in the resulting graph grammar. If a procedure uses the value of one of its arguments in a linear manner, then the reference to that value will always be a direct connection—there will be no need for a tree of `Copy` vertices. In this way, linearities in the original program are exposed in the linear graph structure model, where we can exploit them during execution.

In chapter 5 Scheme programs will be compiled as described above, and executed by a distributed graph reduction engine. That engine will have no understanding of what the various vertex types meant to us in this chapter. In a sense it isn't necessary to have an understanding of this chapter in order to understand chapter 5, but you cannot truly appreciate the action without it. When the distributed graph reduction engine decides to transport a subgraph containing a vertex of type `Call 2` from one network location to another, you may see why this act is good graph reduction strategy, but if you understand this chapter, the act acquires additional significance as an example of a remote procedure call.

[7] gcc

Chapter 4

Maintaining Connections Across the Network

This chapter describes the **link** abstraction and the network protocol that implements it. The distributed graph reduction engine is made up of a collection of **agents** that each hold a subset of the vertices in the working graph. Links are used to maintain connections between vertices that are held by different agents. In chapter 5 we will use the mechanism described in this chapter to support the execution of the code generated in chapter 3.[1]

A link has two **ends**, which can travel independently from agent to agent. A agent that is currently in possession of one of the two ends of a link can inquire about the most recently known location of the *other* end. A simple network protocol guarantees that each end is promptly informed about changes in the location of the other.

The migration of a subgraph from one agent to another will typically cause multiple link ends to pass from agent to agent over the network. As we shall see in chapter 5, this is the *only* significant cost of subgraph migration. Thus the more expensive it is to move link ends, the more expensive it will be to migrate subgraphs. It is therefore worthwhile to work at making the implementation of links as light-weight as possible.

Fortunately links are able to meet the goal of being light-weight because they implement *linear* references. A nonlinear reference mechanism has to worry about a number of things that linear references avoid. Nonlinear references can be duplicated and stored in an arbitrary number of different locations. The target of a nonlinear reference must be prepared to handle multiple requests from

[1] In order to better understand how the link abstraction fits in to the rest of the system, the reader may find it helpful at this point to preview figure 5.1 on page 88, which shows how the run-time modules support each other. The link module appears fairly low down, supported only by the reliable message service.

reference holders, and must continue to exist until some separate mechanism determines that there are no more outstanding references. Any implementation of nonlinear references that cross a network must take these possibilities into account—see [YTR+87] for an example of the consequences.

In contrast, linear references cannot be duplicated. The entity at the other end of a linear reference need only be prepared to handle a single request and can then cease to exist. Support for linear references that cross a network only needs to worry about the whereabouts of *one* remote entity. If that entity ever becomes local, then all network resources devoted to maintaining the linear reference can be easily discarded.

4.1 The contract of a link

Links capture the essence of the requirements for supporting cross-network connections in a simple abstraction. The following five procedures define the complete interface to the abstraction. Note that these procedures are all invoked by some particular agent—that calling agent is an implicit argument to them all.

`create_link()` ⇒ *Link* procedure

> Creates a new link. The calling agent will be given possession of both ends of the new link.

`destroy_link(`*Link*`)` procedure

> Destroys *Link*. The calling agent must possess both ends of *Link* before calling `destroy_link`.

`pick_up_link(`*Link*`, `*Agent*`)` ⇒ *Descriptor* procedure

> Starts the process of moving one end of *Link* from the calling agent to *Agent*. The *Descriptor* is an identifier that can be presented to `put_down_link` (see below) to complete the move. The calling agent must possess at least one of the two ends of *Link* before calling `pick_up_link`, and after the call it will possess one fewer.

`put_down_link(`*Descriptor*`)` ⇒ *Link* procedure

> Finishes the process of moving a link. The calling agent now has possession of an additional end of *Link*. The calling agent must be the agent specified in the call to `pick_up_link` that created *Descriptor*.

`query_link(`*Link*`)` ⇒ *Agent* or `NULL` procedure

> Queries *Link* about the location of its other end. If the calling agent possesses both ends of the link, then `NULL` is returned. The calling agent must possess at least one of the ends of *Link* before calling `query_link`.

A **descriptor** can be thought of as the portable representation of a link end. From the point of view of the users of this interface, the only purpose served by a descriptor is to prevent the caller from confusing multiple link ends that are in transit at the same time. (Another way to accomplish the same thing would be to arrange for link ends to arrive in the same order that they were transmitted.)

From within the link layer, descriptors serve a devious additional purpose. The only thing the user of the link abstraction can do with a descriptor is carry it to the destination agent and call `put_down_link`. The link layer takes advantage of this by making information that it wants to transmit to the destination agent *part of the descriptor itself*. This hack allows the link layer to avoid sending its own reliable messages in many (but not all) situations. In order to make this work properly an agent that calls `pick_up_link` accepts responsibility for reliably delivering the descriptor to the destination agent and calling `put_down_link`. In practice this additional responsibility is no burden.

Note there there is never any need for the calling agent to distinguish between the two ends of a link—for example, by labeling them the "left" and "right" ends. An agent interested in the state of a particular link will either find that it is in possession of both ends, or it will be interested in the location of the *other* end. No other distinguishing characteristics are required.

The two ends of a link can be thought of as two virtual tokens. These tokens are passed from agent to agent as part of the descriptors. At any given moment, each token exists in exactly one place in the network: either it is in the possession of some agent, or it is part of a descriptor. The contract of the link layer is to guarantee that:

- Neither of these tokens will be duplicated or lost. (This is easy to accomplish as long as descriptors are delivered reliably.)

- If one token stays immobile in the possession of an agent, then that agent will receive prompt updates about the location of the other token.

- If *both* tokens stay immobile, then after some small number of messages (perhaps two or three) have journeyed through the network, each possessing agent will learn the identity of the other.

- If neither token is moving, no network traffic will be required to maintain the link.

- An agent that does not currently possess either token from a given link will be able to easily reclaim all local resources devoted to maintaining that link—even though the link continues to exist, and may even pass through this agent again in the future.

- When a link is destroyed (via a call to `destroy_link`), all resources on all agents devoted to maintaining that link will be easily reclaimed.

A call to `query_link` does not initiate any network traffic. `query_link` always just immediately returns the most recent local information about the location of the other end of the link. The link module works in the background to update this information as quickly as possible, but there is no guarantee that the claimed location of the other token reflects current reality. The other token may have recently departed from the claimed location, or alternatively it may still be part of a descriptor en route to the claimed location.[2]

Higher level modules must be designed to take the limitations of `query_link`'s answers into account. In section 5.3, when graph structure migration is described, this fact will play an important role.

4.2 The link maintenance protocol

In this section I will describe the link maintenance protocol in detail. This protocol meets all the requirements set forth in the previous section, and is clearly good enough to demonstrate that cross-network connections can be kept light-weight. There are still some problems with this protocol, and my intuition is that even better protocols are possible.

There are two desires that motivate the design of this protocol. First, there is the desire to quickly propagate changes in the location of one end of a link to the agent that holds the other end. Second, there is the desire to easily reclaim resources devoted to links whose ends are no longer present at a given agent.

The most obvious kind of protocol, where agents forward messages addressed to link ends that have recently departed, is unsuitable on both counts. First, if both ends of a link are hopping rapidly from agent to agent, then it can take an arbitrary number of forwarding steps before the most up-to-date information about one end can arrive at the other end. Second, it is difficult for an agent to know when it is safe to forget what it knows about a link if the protocol relies on agents to provide forwarding.

The protocol described here solves these problems by using a fixed **home** agent for each link that is always kept informed about the locations of the two ends. The home agent of a link will be the agent who called `create_link` to forge the link in the first place. The home agent receives status reports from the two ends whenever they move. Occasionally the home agent will notice that the two ends have potentially lost track of each other, in which case it will send messages to correct the situation. This guarantees that an agent holding one end of a link will learn the true location of the other end of the link after at most a single forwarding step. This also allows any agent other than the home

[2] It is in fact possible for an agent to call `query_link` and find that the other end of the link is believed to be possessed by the calling agent itself, even though only one token is actually resident locally. This can happen if the token for the other end is in transit and was overtaken by some of the messages that the link layer itself exchanges—once the token for the other end arrives, calls to `query_link` will return `NULL`, as expected.

agent to discard all knowledge of any link whose ends are elsewhere.

Relying on a third party home agent does have its disadvantages. If the home agent becomes inaccessible for some reason, then the two ends of a link may lose track of each other, even through they may be within easy reach of each other. Even if the home agent remains accessible, the two ends may wander far away from home, so that when the home agent's services are required a lot of long distance network traffic takes place to correct what should be a purely local problem.

These are not particularly bad problems for very short-lived links, but for links that last more that a few seconds it would be nice to shift the home agent's responsibilities from the original creator to one of the two agents currently holding an end of the link. Although the current run-time system does not implement it, I will suggest a technique that addresses this problem below.

4.2.1 Link maintenance data structures

The link layer is built on top of a reliable message layer, which assigns a 64-bit **Agent-Id** to each agent. Each link is assigned a unique **Link-Id** when `create_link` is called. The 95-bit Link-Id consists of the 64-bit Agent-Id of the home agent (who called `create_link`), plus a 31-bit extension generated by the home agent.[3] This makes it easy to obtain the Agent-Id of the home agent of a link given an arbitrary Link-Id.

Inside the link layer, the two ends *are* distinguished. They are called the **up** and **down** ends of the link. (The up end prefers to travel, although nothing takes advantage of this fact.)

Each agent maintains a table of **link records**, keyed by Link-Id. A link record contains the following fields:

`id`

The Link-Id of this link. Recall that this includes the Agent-Id of the home agent for the link.

`up_agent`

The Agent-Id of the agent believed to be in possession of the up end of this link.

`down_agent`

The Agent-Id of the agent believed to be in possession of the down end of this link.

`up_sequence`

A 16-bit sequence number associated with the information currently stored in `up_agent`. The way sequence numbers are used is described below.

[3]The issue of what to do when an agent runs out of 31-bit extensions is not addressed in this implementation. Link-Ids could be reused after a suitable timeout.

`down_sequence`
: A 16-bit sequence number associated with the information currently stored in `down_agent`.

`up_flag`
: This flag is set if the up end of the link is actually held by the local agent. If this bit is set, then `up_agent` will necessarily contain the Agent-Id of the local agent. (The converse is *not* true.)

`down_flag`
: This flag is set if the down end of the link is actually held by the local agent. If this bit is set, then `down_agent` will necessarily contain the Agent-Id of the local agent. (`up_flag` and `down_flag` record the presence of the tokens discussed in section 4.1.)

`deleted_flag`
: This flag is set if the link has been destroyed.

4.2.2 Link layer communication

The link layer relies on other layers to transport descriptors, but it also needs to exchange messages within itself. Thus the link module uses the same reliable message service used by the migration routines that will be described in section 5.3. This introduces some unnecessary communication in some circumstances since the link layer doesn't actually need all the power of reliable messages, but it has the virtue of simplifying the link layer—making it easier to implement, debug and explain.

4.2.3 Creating a new link

When an agent calls `create_link`, a new Link-Id is generated (with the calling agent as home agent), and a new link record for that link is added to the calling agent's link record table. In that new record the `up_agent` and `down_agent` are set to be the calling agent's own Agent-Id, the `up_flag` and `down_flag` are set, the `deleted_flag` is clear, and the `up_sequence` and `down_sequence` are initialized to 0.

4.2.4 Destroying a link

When an agent calls `destroy_link` the system first checks to be sure that both `up_flag` and `down_flag` are set, otherwise the calling agent does not have permission to destroy the link. Then, if the calling agent is not itself the home agent, a message is dispatched to the home agent informing it of the demise of the link.

Both the destroying agent and the home agent set the `deleted_flag` in their records for that link. They could immediately discard these records, but as we shall see below, there are advantages to holding on to this information for a short time.

4.2.5 Moving one end of a link

When an agent calls `pick_up_link` the system first checks to be sure that one of `up_flag` or `down_flag` is set, otherwise the calling agent does not have permission to move the link. The system then picks one of the ends that the calling agent possesses—without loss of generality we can assume that this is the up end—and modifies it as follows:

- The `up_flag` is cleared.
- The Agent-Id `up_agent` is set to be the destination where the caller is planning on sending the descriptor.
- The number `up_sequence` is incremented.

A copy of this modified link record is sent in a reliable message to the link's home agent, a second copy is sent via reliable message to the agent named in `down_agent`, and a third copy is made part of the descriptor that is returned by `pick_up_link` (which will eventually be delivered to the destination agent now named in `up_agent`). The `up_flag` and `down_flag` in these copies are cleared, except the `up_flag` in the descriptor is set in order to tell the recipient which end it is getting.

The effect of this is to insure that the following four agents all eventually learn about the new state of the link: (1) the caller of `pick_up_link`, who just packed the up end into a descriptor; (2) the destination agent, who will soon receive the up end inside that descriptor; (3) the agent believed to be holding the down end; and (4) the home agent.

Of course some of these agents may actually be the same, in which case we avoid sending duplicate messages. For example, it is common for the caller of `pick_up_link`, the agent holding the other end of the link, and the home agent, to all be the same agent (this will always be the case for a link newly created by `create_link`). In this case, only the copy of the new link record transmitted as part of the descriptor need actually be sent.

Whenever any agent receives a link record, either in a descriptor, or in a message sent within the link layer, it merges the new information in this **update record** with whatever information it has stored in its own link table. If it does not currently have a link record for the link in question, it simply copies the update record into its table. If it already has a record for the link, then it separately merges the up and down halves by retaining the information with the larger sequence number.

If the merging agent is the link's home agent, then there are some additional considerations. First, if the home agent discovers that it no longer has a local record for that link, then in the new record the `deleted_flag` will be set. This ensures that destruction of a link is never forgotten, because its destruction is the only way its link record can ever vanish from its home agent's link table. Thus at any point in the future an agent that needs to know if a given link has been destroyed can always ask the link's home agent. Certain rare cases of link record reclamation, described below, make use of this ability.

Second, recall that the home agent's main function is to guard against the possibility that the two ends of the link have lost track of each other. For this reason the home agent compares the new local record with both the update record and the old local record. If the new record differs from *both* of the others, then it is possible that the two ends of the link have become separated. In that case, the two agents named in the `up_agent` and `down_agent` of the new local record are each sent a copy of the record. Those two updates will themselves be merged into their recipient's local records using the algorithm just described.

The proof that this algorithm successfully keeps the two ends of a link informed of each other's location is presented in section 4.3.

4.2.6 Reclaiming link records

Periodically, agents scan their link tables and attempt to reclaim link records that no longer serve any useful purpose. A link record can *not* be reclaimed if any of the following are true:

- This is the home agent of the link, and the `deleted_flag` is not set.

  The home agent is depended upon to never forget anything it has ever been told about its own links (until more recent information arrives).

- One of `up_flag` or `down_flag` is set, and the `deleted_flag` is not set.

  Clearly to reclaim such a record would be to permanently lose one of the two ends of the link.

- The record has been recently used.

  This condition is not necessary to the correctness of the algorithm, but it does improve performance in a couple of cases discussed below. Clearly there is little to lose by retaining useful information.[4]

Otherwise the record is a candidate for reclamation.

[4] The current implementation won't reclaim a record that has been used in the last 30 seconds. There are many alternatives to this simple approach. For example, agents could wait until their link record table grows beyond a certain size, and then reclaim enough of the oldest records to reduce that size by a given amount.

If the `deleted_flag` is set in a reclamation candidate, then the record is reclaimed. If the `deleted_flag` is *not* set in a reclamation candidate, then, because of the three checks listed above, we know that:

- This is not the home agent.
- Neither the `up_flag` nor the `down_flag` is set.
- The record has not been recently used.

In this case the agent compares the `up_agent` and `down_agent` in the record with its own Agent-Id. If neither matches, then again the record is reclaimed. However, if one does match, then it is possible that the actual end of the link is en route to this agent, and that this record represents useful information that will be needed once that end arrives. This might happen if the home agent detected some confusion and sent update records to the two ends—such an update might actually arrive in advance of the end itself. (Of course this is highly unlikely to be the case if the record has been sitting around untouched for a long time, which is one reason not to reclaim records until they have gotten a little stale.)

So in this one case, the agent must actually correspond with the link's home agent in order to determine whether or not it can reclaim the record. It does this by sending a message to the home agent requesting that an update message be sent back. When that update arrives it will either reveal that the link has been destroyed, or it will change the `up_agent` and `down_agent` so that they no longer refer to the local agent, or *perhaps* it will confirm that the record really is up-to-date information, in which case the agent can only hold on to the record and wait for it to prove useful.

The safety of this algorithm is proved in the next section.

4.3 Proof of correctness

In this section, I will prove that the link maintenance protocol described in the previous section is correct. There are two things that must be proved:

- The two ends of a link can never permanently lose sight of each other. More precisely, if an agent receives a link end and holds on to it, it is guaranteed to eventually learn about the location of the other end.
- The algorithm for link record reclamation never discards any useful information.

I will start by proving the first property under the assumption that agents never discard *any* link records.

Whenever either end of a link changes its location, it always dispatches an update record reflecting that change in a reliable message to the link's home agent. Thus, the home agent always eventually learns where the ends are located. The algorithm employed by the home agent guarantees that the following invariant is maintained:

> An update containing the same `up_sequence`, `up_agent`, `down_sequence` and `down_agent` as are contained in the home agent's link record has been sent to both the `up_agent` and the `down_agent` named there.

In other words, the home agent ensures that *precisely* what it currently knows has been sent, in a single message, to each of the two agents that it believes need that information.

The home agent can usually maintain this invariant without doing any work at all. As described in section 4.2.5, after the home agent performs the merge algorithm, it compares the new local record with the old local record and the update record. If the new local record matches the old local record, then the invariant remains true because the local record hasn't changed. If the new local record matches the received update record, then the invariant remains true because the agent that sent the update also sent copies of that record to the two link end holders—it already did the work necessary to maintain the invariant.

Only in the case where the new local record differs from both the inputs to the merge will the home agent have to take any action in order to preserve the invariant. In that case, it simply generates the two necessary updates itself. (This case can only happen if the information in one half of the update is strictly newer than what the old local record contained, while the information in the other half is strictly older. This, in turn, can only happen in the presumably rare case where the two link ends move at almost exactly the same time.)

Since all location changes are sent to the home agent, maintaining this invariant guarantees that the ends never completely lose touch with each other. If an agent holds on to a link end for long enough, then eventually the home agent will learn both where this end is, and where the other end is. At that time, the invariant ensures that *somebody* will have sent the agent an update containing what the home agent knows.

This completes the proof that the two ends cannot lose each other. This proof assumed that agents never forget anything. Now we have to worry that the link record reclamation process might foul things up by discarding useful information.

There are two cases to consider. In the easy case the `deleted_flag` is set in the record. This can only happen if the link has been destroyed, in which case the only information that needs to be preserved about the link is the news of its destruction. We have already seen that the home agent is always able to reconstruct this fact given a Link-Id.

The second case of link record reclamation occurs when (1) the reclaiming agent is not the home agent of the link and (2) the `up_agent` and `down_agent` of the record do not contain the Agent-Id of the reclaiming agent. So the reclaiming agent is neither the home agent, nor is it currently holding either end of the link. Such an agent will never again need to know anything about the link in question unless some other agent decides to send one of the ends back to it. In that case, we know that the home agent will work to ensure that the reclaiming agent will be sent a single update record that contains *everything* it needs to know. So our only concern is to avoid reclaiming the current local record *after* that vital update has arrived.

That vital update, however, will always mention the reclaiming agent as either the most recent `up_agent` or `down_agent`, and the corresponding sequence number will be larger than any sequence number yet associated with the location of that end. If such an update had arrived locally, the merge algorithm would therefore have preserved that information. So since neither the `up_agent` nor the `down_agent` mention the reclaiming agent, such an update, if it exists at all, has not yet arrived. And it is therefore safe to reclaim the local link record.

4.4 Examples

In order to give the reader a better understanding of how the link maintenance protocol behaves in practice, this section contains several examples of it in action.

Throughout these examples H will be the home agent of the example link and A, B and C will be other agents. Initially H calls `create_link` and is given possession of both ends of a newly forged link. A link record is placed in H's link record table that describes this situation.

As there will be a lot of link records flying around in the descriptions that follow, it will help to have a concise notation for them. Link records will be written as follows:

$$\langle \texttt{up\_sequence}, \texttt{up\_agent}, \texttt{down\_sequence}, \texttt{down\_agent}, \mathit{Flags} \rangle$$

where *Flags* is a subset of $\{\texttt{up}, \texttt{down}, \texttt{deleted}\}$. For example, immediately after H calls `create_link`, the state of the world is:

$$H \text{ is holding:} \quad \langle 0, H, 0, H, \{\texttt{up}, \texttt{down}\} \rangle$$

Note that we don't need to specify which link the record describes, since all the examples only concern a single link.

4.4.1 There and back again

In the simplest possible case, H immediately calls `destroy_link`. That case is neither very interesting nor very likely, so we pass immediately to the second simplest case, where H sends one end of the link to A, who immediately returns it.

First, H calls `pick_up_link`, passing it the link and the Agent-Id of its intended destination, A. Then it sends the resulting descriptor to A. The resulting state:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ \text{In a descriptor bound for } A\text{:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

Note that up end has been chosen to move from H to A, and the sequence number in the up half of the record has been incremented to reflect this fact. Since H is the home agent, and since H is still holding the down end of the link, no additional update records were generated.

When the descriptor arrives at A, A calls `put_down_link`. This creates a new link record at A:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

At this point, if either A or H calls `query_link`, it will be told that the other agent is holding the other end of the link.

Now A decides to send its end back to H. A calls `pick_up_link` and sends the resulting descriptor back to H:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 2, H, 0, H, \{\}\rangle \\ \text{In a descriptor bound for } H\text{:} & \langle 2, H, 0, H, \{\texttt{up}\}\rangle \end{array}$$

Since A is sending the end back to the home agent, who it also believes to be holding the other end, again no additional updates were generated.

At this point a call to `query_link` by H would still claim that A is holding the other end, even though this is no longer the case. But this situation only persists until the descriptor arrives at H and is merged in with the existing record:

$$\begin{array}{rl} H \text{ is holding:} & \langle 2, H, 0, H, \{\texttt{up}, \texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 2, H, 0, H, \{\}\rangle \end{array}$$

Now H is holding both ends, so it can call `destroy_link`. Since H is the home agent, an update is not needed to inform the home agent of the destruction. The result:

$$\begin{array}{rl} H \text{ is holding:} & \langle 2, H, 0, H, \{\texttt{up}, \texttt{down}, \texttt{deleted}\}\rangle \\ A \text{ is holding:} & \langle 2, H, 0, H, \{\}\rangle \end{array}$$

The algorithm for link record reclamation will recognize that both of these records may be reclaimed whenever the agents that hold them find it convenient. (In fact, A could have reclaimed the record that it holds any time after it send the end back to H.)

Note that in this simple case, the link layer did not send *any* messages at all. All network traffic was contained in descriptors carried in messages sent by other layers. Frequently the link layer can get away with being completely parasitic on the messages sent by other layers.

4.4.2 Follow the leader

Now let us return to the state just after A called `put_down_link`:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

Now let us suppose that instead of A sending its end back to H, H sends the other end on to A. So H calls `pick_up_link` and sends the descriptor off to A:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, A, \{\}\rangle \\ \text{In a descriptor bound for } A\text{:} & \langle 1, A, 1, A, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

A merges in the descriptor:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, A, \{\}\rangle \\ A \text{ is holding:} & \langle 1, A, 1, A, \{\texttt{up}, \texttt{down}\}\rangle \end{array}$$

Now suppose that this time A calls `destroy_link`. This requires A to send an update record in a link layer message to H to inform it of the destruction of the link:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, A, \{\}\rangle \\ A \text{ is holding:} & \langle 1, A, 1, A, \{\texttt{up}, \texttt{down}, \texttt{deleted}\}\rangle \\ \text{In an update for } H\text{:} & \langle 1, A, 1, A, \{\texttt{deleted}\}\rangle \end{array}$$

And finally after H receives the news:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, A, \{\texttt{deleted}\}\rangle \\ A \text{ is holding:} & \langle 1, A, 1, A, \{\texttt{up}, \texttt{down}, \texttt{deleted}\}\rangle \end{array}$$

Again we have arrived at a state where either agent can reclaim its record of the link whenever it desires.

In this case there was only one link layer message sent, due to the fact that the link was destroyed somewhere other than at its own home agent.

4.4.3 Wandering around away from home

Again let us return to the state just after A called `put_down_link`:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

This time, suppose that A decides to pass its end of the link on to C. So A calls `pick_up_link` and sends the descriptor off to C:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\ \text{In a descriptor bound for } C\text{:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle \\ \text{In an update for } H\text{:} & \langle 2, C, 0, H, \{\}\rangle \end{array}$$

Notice that this required A to send an update record back to H both because H is the home agent and because A believes H is holding the other end. After H processes that update, and after C merges the descriptor:

$$\begin{array}{rl} H \text{ is holding:} & \langle 2, C, 0, H, \{\texttt{down}\}\rangle \\ A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\ C \text{ is holding:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle \end{array}$$

At this point A is free to reclaim its record of the link and both H and C are aware of who is holding the other end of the link.

This example demonstrates that as long as one end of the link stays at home, the price of moving the other end is a single link layer message.

4.4.4 Everybody leaves home

Now let us consider what happens when both ends leave home for separate destinations. After sending one end to A, suppose H calls `pick_up_link` again and sends that descriptor to B:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, B, \{\}\rangle \\ \text{In a descriptor bound for } B\text{:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \\ \text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\ A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle \end{array}$$

An update was dispatched to A to keep it informed of the location of the other end. After A processes that update, and after B merges the descriptor:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, B, \{\}\rangle \\ A \text{ is holding:} & \langle 1, A, 1, B, \{\texttt{up}\}\rangle \\ B \text{ is holding:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \end{array}$$

Now suppose A wants to pass its end on to C. It calls `pick_up_link` and sends the descriptor:

$$\begin{array}{rl} H \text{ is holding:} & \langle 1, A, 1, B, \{\}\rangle \\ A \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\ \text{In a descriptor bound for } C\text{:} & \langle 2, C, 1, B, \{\texttt{up}\}\rangle \\ \text{In an update for } H\text{:} & \langle 2, C, 1, B, \{\}\rangle \\ \text{In an update for } B\text{:} & \langle 2, C, 1, B, \{\}\rangle \\ B \text{ is holding:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \end{array}$$

Two update messages were generated, one for the home agent H, and one for B, the agent believed to be holding the other end. Once everything settles down:

$$\begin{array}{rl} H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\ A \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\ C \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{up}\}\rangle \\ B \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{down}\}\rangle \end{array}$$

A is free to reclaim its record, and B and C each know that the other holds the other end of the link.

So we see that when the second end leaves the home agent, one link layer message is required, and whenever either end moves from there on in, two link layer messages are required.

4.4.5 Confusion reigns

In the previous example, an interesting case occurs if A passes its end of the link on to C *before* it receives word that the other end has traveled from H to B. We return to the state just after H takes that action:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 1, A, 1, B, \{\}\rangle \\
\text{In a descriptor bound for } B\text{:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 1, A, 0, H, \{\texttt{up}\}\rangle
\end{array}$$

Now this time, before any messages are delivered, A calls `pick_up_link` and sends that descriptor to C:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 1, A, 1, B, \{\}\rangle \\
\text{In a descriptor bound for } B\text{:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\
\text{In a descriptor bound for } C\text{:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle \\
\text{In an update for } H\text{:} & \langle 2, C, 0, H, \{\}\rangle
\end{array}$$

Since A is unaware that the other end is on its way to B, it only generated an update for H. Let us suppose that that update now arrives at H and is merged:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } B\text{:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } C\text{:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In a descriptor bound for } B\text{:} & \langle 1, A, 1, B, \{\texttt{down}\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\
\text{In a descriptor bound for } C\text{:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle
\end{array}$$

The merge resulted in a record that was different from both the inputs, so H generated updates for B and C. Note that A is currently free to reclaim its record.

So far, no messages have arrived at either B or C. At B, suppose the descriptor arrives first, followed by the update. The result:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } C\text{:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\
\text{In a descriptor bound for } C\text{:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle \\
B \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{down}\}\rangle
\end{array}$$

B now believes that C is holding the other end, even though it hasn't actually arrived there yet. (Before the update arrived, a call to `query_link` at B would have claimed that A held the other end, even though it had already departed.)

At C, let us suppose that the update from H arrives first. The result:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\
\text{In a descriptor bound for } C\text{:} & \langle 2, C, 0, H, \{\texttt{up}\}\rangle \\
B \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{down}\}\rangle \\
C \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle
\end{array}$$

Notice that at this point, the record at C is a candidate for reclamation, but since C itself appears in the record, C must ask H for another update if it gets anxious about that record—and in this situation another update from H will simply reassure C that the record it holds really is important. But this is unlikely to happen, because the descriptor sent to C by A will almost certainly arrive long before reclamation becomes an issue. The result of merging that descriptor will be:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
\text{In an update for } A\text{:} & \langle 1, A, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 0, H, \{\}\rangle \\
B \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{down}\}\rangle \\
C \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{up}\}\rangle
\end{array}$$

So now B and C know all about each other.

Only the update destined for A remains to be delivered. Since A is unlikely to get around to reclaiming the record it is currently holding before that happens, the results after delivery and merging will be:

$$\begin{array}{rl}
H \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
A \text{ is holding:} & \langle 2, C, 1, B, \{\}\rangle \\
B \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{down}\}\rangle \\
C \text{ is holding:} & \langle 2, C, 1, B, \{\texttt{up}\}\rangle
\end{array}$$

The updated record that A is now holding is still one that can be reclaimed at will. Notice that if A had reclaimed its previous record before the update was delivered, then that update would have created a new record in which A itself was mentioned. In order to reclaim that new record A would have had to first request another update from H. This demonstrates that there are benefits to not reclaiming link records too quickly.

The difference between this example and the one in the previous section was that in this case the two ends briefly lost track of each other. We saw that it took two additional link layer messages to correct the situation.

We can also see that the worst possible case, in terms of the number of link layer messages needed to support the protocol, is when both ends have traveled away from the home agent, and then they both move again nearly simultaneously. In that case, six link layer message will be sent: two each from the two sending agents (one to the home agent, and one to the other sending agent), and two more from the home agent to the new end holders.

4.5 Analysis and possible improvements

In section 4.1, I listed the requirements that the link maintenance protocol was designed to meet. The degree to which those requirements have been satisfied can now be evaluated.

The basic integrity requirement, that neither of the two ends be duplicated or lost, was never at issue. As long as descriptors are delivered in reliable messages, this is easy to achieve. Similarly, there was never any question that as long as neither end of a link was moving, no network traffic would be needed to maintain the link.

In order to evaluate how well the requirements have been met that agents receive prompt updates about locations, consider the case of an agent, A, which has just obtained one end of a link from a descriptor. How long must A wait before it learns the location of the other end? A knows that at the time the transmitting agent called `pick_up_link`, that agent also sent an update to the home agent to announce that A was the new location. Once that update arrives, the home agent will know where to send future updates about the location of the other end. Thus, even if the other end never learns about A, the updates it sends to the home agent will be relayed on to A.

So after an initial short wait, A will start receiving current information that has passed through at most a single third party. This might not be a particularly quick way to keep in touch, especially if the home agent is located very far away, but it is much better than a scheme where an unbounded number of forwarding agents may intervene.

Finally, it was required that link record reclamation be easy. Ideally, any agent that is not currently holding either of the two ends of a link should be able to reclaim its record of that link whenever it wants—without consulting any other agents. This protocol achieves that goal, with two exceptions.

First, as we saw at the end of the last example, in some very rare cases an extremely old update can arrive at an agent long after that agent had discarded its previous record of the same link. The record created by such an update cannot be reclaimed without probing the home agent for confirmation. The longer agents hold on to old records, the more unlikely this case becomes, so the cost here can be made negligible.[5]

Second, the home agent must preserve its link record even after both ends of the link have left home. This is another instance where relying on a fixed third party agent is a bit troublesome.

One important special case performs particularly well. If there are only two agents ever involved in the history of a link, then no link layer messages will be required at all. All the correspondence between those agents about that link can be carried within the descriptors exchanged as part of the higher level protocols. (The first two examples above demonstrate this.)

This case is important because it occurs whenever the higher level is behaving as it would for a remote procedure call. That is, if two agents pass subgraphs back and forth that represent procedure calls, call arguments, procedure returns and returned values, then the links created to support the process will all be of this special kind.

The obvious way to improve this protocol is to provide some way to move the home agent. While the home agent continues to hold one of the two ends of the link, there isn't much room for improvement, but as soon as the home agent becomes a third party, we find ourselves in a state where many things would perform better if only the home agent could be moved to where one of the ends was located.

In fact, we can achieve the effect of moving the home agent by discarding the old link and replacing it with a new one, forged by the agent that would be a better home. This optimization is not done in the current implementation, but it would be easy to add.

For example, if A and B are holding the ends of a link forged by H, and A decides that it would be a better home agent for such a link, A can call **`create_link`** to forge a new link, and then send B a message containing two descriptors: one descriptor contains A's end of the *old* link, and the other contains one of the two ends of the *new* link. When B receives the message it checks to see if it still has the other end of the old link. If it does, B calls **`destroy_link`** to dispose of the old link and starts to use the enclosed end of the new link. If the other end of the old link has departed, then B packs the two ends back up and sends the message off to chase after it.

This technique for home agent relocation operates as a separate layer on top of the link layer. The link layer is already built on top of a reliable message layer. The resulting structure is three levels deep, all to accomplish the simple job of link maintenance.

It is clear that the bottom two layers could profit by being combined into a single layer. For example, there are occasions where messages sitting in an

[5] It may even be possible to eliminate this case entirely using an argument based on maximum packet lifetimes.

agent's reliable message layer retransmission queue can be abandoned because the link maintenance layer at that agent now has more up-to-date information to transmit. The fact that both the reliable message layer and the link layer maintain their own sequence numbers suggests how closely related these two layers really are. My intuition is that using a third layer for home agent relocation is also a mismodularization; I believe that a better protocol can be designed that combines all three of these layers into a single unified protocol.

4.6 Summary

In this chapter I described the link abstraction which implements cross-network linear naming, and I demonstrated that links can be implemented cheaply. Linearity is important in keeping links cheap because it guarantees that each end of a link only has to think about *one* thing: the location of the other end.

In the next chapter links will be used to support the cross-network connections necessary for distributed linear graph reduction. The fact that links are cheap will mean that linear graph structure can be easily moved from agent to agent.

Chapter 5

Distributed Execution

This chapter describes the distributed linear graph reduction engine. It describes the behavior of the individual **agents** that hold the vertices that make up the working graph. Agents exchange vertices through the network and they apply methods to the subgraphs composed of the vertices they currently possess.

Agents decide which vertices to migrate, when to migrate them, and who to migrate them to, by using a few simple heuristics. These heuristics are effective because the lifetime cumulative cost of maintaining a cross-network link is both predictable and small. The same could not be said of traditional nonlinear, pointer-like reference mechanisms.

5.1 Run-time modules

Figure 5.1 depicts how the modules that comprise the run-time system support each other. At the very bottom lies a module that maintains data structures that represent the local part of the working graph, and a module that provides a reliable message service for interagent communication. Other modules are constructed on top of those two, and at the top sits a scheduler that decides what to do next.

I will now briefly describe each module and how it relates to the other modules.

Reliable Messages. Agents communicate with each other using a simple reliable message service. This service is constructed on top of the Internet's UDP protocol [Pos80], although any unreliable datagram service would serve as well. The reliable message service maintains its own sequence numbers, timers, retransmission queues, and delivery queues. It encapsulates its messages and acknowledgments within UDP datagrams. Each reliable message is transported

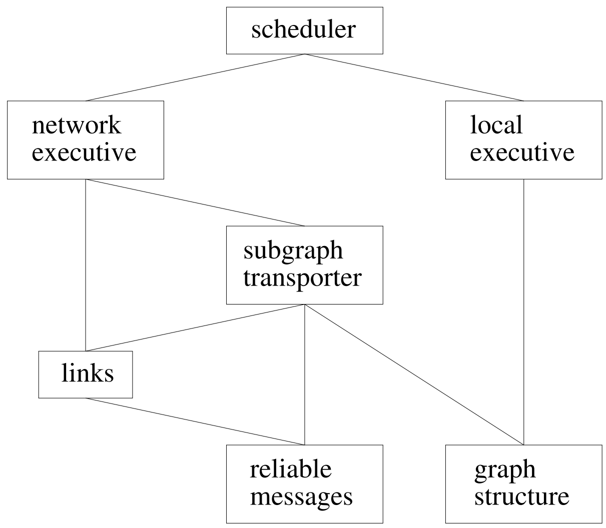

Figure 5.1: Run-time modules

in one datagram sent to the destination and acknowledged in a second datagram sent back to the source.

The reliable message service uses a 64-bit **Agent-Id** as the address of an agent.

Links. The abstraction of a **link** is used to maintain connections over the network. Chapter 4 described how links are implemented in detail. Recall that every link has exactly two ends somewhere in the network, and that if an agent is holding one of the two ends, then it will be kept informed about the current location of the *other* end. In order to accomplish this, the link module uses the reliable message service to exchange messages with link modules running on other agents.

Graph Structure. The graph structure module is partly a storage manager. It contains routines that allocate and free the blocks of storage used to represent vertices (as described in chapter 3). This module is also charged with maintaining two queues: The **reduction queue** contains redexes that are waiting to be reduced locally, and the **migration queue** contains redexes that are unable to reduce because one of the two vertices is held by a remote agent.

Compiled method procedures (the results of the compilation procedure described in chapter 3) are distributed to all agents. These procedures use utilities in the graph structure module to alter the working graph. These utilities are careful to delete entries from either queue if they concern vertices that have been removed, and to add appropriate new queue entries.

The tagged values used for connections make it easy to spot redexes whenever a new connection is made. Code that constructs new graph structure calls the procedure `connect_dispatch` to make new connections between vertices, passing it the two tagged values that represent the terminals that are to be connected. By extracting the two tags, concatenating them, and looking the result up in a table, `connect_dispatch` can quickly dispatch to an appropriate routine that knows exactly how to make the desired connection. For example, when connecting the `target` terminal of a `Car` operation to the handle of a `Cons` object, `connect_dispatch` can tell from the two tags alone that a method[1] applies whenever those two terminals are connected, so it makes an entry in the reduction queue.

A special tag is used to identify connections to vertices held by agents elsewhere in the network. In this case, the rest of the tagged value contains the address of (the local part of) a link. Given a link, the system can determine the Agent-Id of the agent holding the vertex at the other end of the connection, and the tag that the value *would have* if it were a connection to a locally held vertex (call this the "true tag").

When `connect_dispatch` is given a value tagged as a connection to a remote vertex, it dispatches to a routine that pulls the true tag out of the link and checks to see if some method would apply if the two vertices were held by the same agent. If so, then we say that the redex is **blocked** by the network. Blocked redexes are stored in the migration queue.

Subgraph Transporter. The subgraph transporter can move arbitrary subgraphs from agent to agent. Subgraph transporter modules running on different agents exchange messages using the reliable message service. Such messages use a simple language to describe graph structure that was recently disassembled by the sender, and that should be reassembled by the receiver. These messages also contain links[2] that specify how the subgraph is joined to the rest of the working graph.

Before sending a subgraph, the transporter calls the graph structure module to disassemble the structure to be sent, and it calls the link module to create links to support newly created interagent connections. After receiving a subgraph, the transporter calls the graph structure module to reassemble it, and the link module to figure out how to connect it to the rest of the locally held graph.

The subgraph transporter makes no decisions about what vertices to send or where to send them. It is perfectly capable of picking up an arbitrary vertex from the middle of the locally held graph, and sending it to an arbitrary agent. It

[1] Specifically, the method on page 38.

[2] Carried in descriptors, as described in chapter 4.

is *not* necessary for the transported graph structure to be connected to anything at the destination. Policy decisions about what to move, where to move it, and when to move it are made by the network executive.

Network Executive. The network executive is invoked by the scheduler to drain entries from the migration queue. Recall that entries are made in this queue when the graph structure module discovers that a redex is blocked by the network.

The network executive first checks the local vertex in the blocked redex to see if it is *also* one of the vertices in a redex in the reduction queue. If so, the network executive just puts the blocked redex back on the migration queue. The idea here is to avoid pushing a vertex over the network that local reduction is about to replace with a different vertex—let the local redex run first, and perhaps the need to communicate with the remote agent will be eliminated.

If the local vertex is *not* subject to local reduction, then the network executive calls the link module to determine the Agent-Id of the location of the other vertex in the blocked redex, and then prepares to call the subgraph transporter to move the local vertex to that location.

Since this commits the system to sending a message, the network executive examines the vertices in the neighborhood of the vertex to be transported to see if there might be some benefit in sending some of them along in the same message. The heuristics used by the network executive to select the subgraph to be migrated are described in detail in section 5.3. The selected subgraph is then fired off to its destination.

When the subgraph arrives, the transporter on the destination agent incorporates it into its local graph structure. If the other vertex in the formerly blocked redex is still held by that agent, then soon a local reduction will take place. If not, no harm has been done, the working graph is still intact. If the other vertex has moved elsewhere, then the redex is still blocked on the network, and it becomes the responsibility of the destination agent's network executive, which may or may not decide to send it chasing after that vertex again.

Local Executive. The network executive is invoked by the scheduler to drain entries from the reduction queue. Recall that entries are made in this queue when the graph structure module discovers an unblocked redex. The local executive really has no intelligence of its own. For each entry in the queue it simply calls the compiler generated procedure that implements the reduction.

Scheduler. The scheduler provides three different services, although the module dependency diagram that started this section only shows one of them:

- The scheduler dispatches incoming network messages to the link module or to the subgraph transporter. Different agent's link modules exchange messages to keep every agent that holds one end of a link informed of the location of the other end. The subgraph transporter gets a message when some graph structure arrives from a different agent.

- The scheduler provides timer interrupts to the link module and the reliable message module. The link module uses these occasions to garbage collect its internal data structures (see chapter 4 for details). The reliable message module uses these occasions to retransmit unacknowledged messages.
- And finally, as the module dependency diagram indicates, the scheduler calls the network executive and the local executive to drain their respective queues.

The only nontrivial part of the scheduler is the way it selects entries from the migration and reduction queues for execution. The algorithm it uses ensures that a certain minimum number of local reductions are made between the time a redex first becomes blocked on the network, and the time the network executive first sees it. The idea is to give the graph structure in the neighborhood of the newly blocked redex time to settle down before thinking about moving stuff over the network.

This heuristic works quite well in practice. As we will see in the examples below, it frequently has the effect of delaying remote migration until there are *no* pending local redexes. This is good, because the agent isn't wasting time using the network to achieve some goal when purely local computation might have changed or even eliminated that goal.

5.2 Two examples

Given the brief outline of the run-time system just concluded, some examples of actual execution can now be presented.

5.2.1 The source code

The complete Scheme source code for the first example contains four top-level definitions. First a familiar procedure:

```
(define (fact n)
  (let loop ((a 1) (n n))
    (if (< n 2)
        a
        (loop (* n a) (- n 1)))))
```

The methods generated for this definition were examined in section 3.4.

Second, a somewhat peculiar procedure:

```
(define (force-number n)
  (if (= n 0) n n))
```

The purpose of this procedure may not be immediately obvious. `FORCE-NUMBER` is only needed because of a poor interaction between the way arithmetic is implemented and the way I/O has been left *unimplemented*. When the initial working graph is constructed it will contain a continuation that handles a `Return 1` message by displaying the returned value on the console—even if that value happens to still be a `Future` or a `Sum` vertex.[3] This isn't very informative when one desires a numeric value, so `FORCE-NUMBER` can be used to insure that a value is an actual number. Its result must be a number because a numeric value is required before the comparison with 0 can be completed.

`FORCE-NUMBER` is the first of three kludges revealed in this section. All three are mechanisms introduced to cover up for missing features that a production implementation would necessarily include. In this case, `FORCE-NUMBER` helps compensate for the lack of a true I/O system. A true I/O system would be written in Scheme and would contain methods that printed just the types the user cares to see (such as numbers and `CONS`-cells), and would wait for other types of vertices to metamorphose into those known types.

The third procedure also requires some explanation:

```
(define (remote-fact host n)
  (host (lambda () (fact n))))
```

Clearly, the argument named `HOST` will be a vertex that can be treated as a one-argument procedure. In fact, `HOST` will always behave as if it was the following procedure:

```
(lambda (thunk) (thunk))
```

And so the effect will always be to compute and return the factorial of `N`. The *difference* between calling `FACT` and `REMOTE-FACT` is that before commencing, `REMOTE-FACT` must briefly examine its other argument.

The reason for this curious protocol is the second of this section's three kludges. The problem is that the distributed linear graph reduction engine lacks any motivation to keep the working graph distributed. Given what I have presented so far, an excellent reduction strategy would be for all agents to send every vertex they possess to a single central agent, who would then run the entire problem locally. In most cases this strategy will be optimal in terms of minimizing network traffic and delay. But in a production implementation, some vertex types would be unable to migrate from agent to agent. These vertices would be anchored at a particular agent because they represent a hardware device (printer, disk drive, keyboard, etc.) that is attached to that agent.

For example, keyboard input might be accomplished by having a `Keyboard Input Stream` vertex that behaved as a zero-argument procedure:

[3] Or the a terminal of a `Copy` vertex!

```
(graph (0 1)
  (<Keyboard Input Stream> 2)
  (<Call 0> target:2 tail:0 cont:1))
(graph (0 1)
  (<Keyboard Input Stream> 0)
  (<Return 1> target:1 tail:3 0:2)
  (<Number x> 2)
  (<Drop> 3))
```

where a different x would be returned each time this method was applied, depending upon which key had been pressed next. A `Keyboard Input Stream` vertex refers implicitly to the keyboard of the agent that is holding it, and thus it cannot be moved.

So there are two special aspects to a `Keyboard Input Stream` vertex, (1) some low-level system magic must link it to the actual keyboard, and (2) the run-time system must recognize its immobility. In the current system we reproduce only the second aspect. A `Network Anchor` vertex is a vertex that is anchored to whatever agent is holding it, and that can be treated as a one argument procedure:

```
(graph (0 1 2)
  (<Network Anchor> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Network Anchor> 0)
  (<Call 0> target:2 tail:3 cont:1)
  (<Drop> 3))
```

Since the `Network Anchor` cannot be moved, whenever this method is applicable, the `Call 1` vertex will be forced to travel to the agent that holds the `Network Anchor`.

So the purpose of `REMOTE-FACT` is finally revealed. It expects to be passed a "host" in the form of a `Network Anchor` that is anchored at some specific agent, as well as a number. First it touches the `Network Anchor` by passing it a thunk. This will cause a `Call 1` to travel to the designated agent. If, as is likely, the network executive decides to send the thunk (the argument to the call) along in the same network message, then the evaluation of `(FACT N)` will start execution at that agent. The subsequent computation of $N!$ will tend to stay on that agent due to the inertia created by the strategy of shunning use of the network unless it appears unavoidable. In effect `REMOTE-FACT` performs a remote procedure call to `FACT`.

Fourth and finally, we have the following procedure:

```
(define (main args)
  (or (null? args)
      (force-number
        (+ (remote-fact (car args) 8)
           (remote-fact (car (cdr args)) 9)))))
```

A call to the procedure `MAIN` will be part of the initial working graph. The argument `ARGS` will contain a list of `Network Anchor` vertices that result from parsing the command line. Each command line argument will be a 64-bit Agent-Id. Those agents will be contacted during the construction of the initial working graph, and a `Network Anchor` will be created at each one.

Reading the code for `MAIN` we see that if no agents are specified in the command line, so `ARGS` is empty, the call to `MAIN` simply returns `True`. If two agents are specified, then the first is asked to compute 8!, the second is asked to compute 9!, the results are added, `FORCE-NUMBER` insures that the sum has reduced to a true number, and the answer is returned.

The third of this section's three kludges concerns the need for the case where `ARGS` is empty. After the value `True` has been displayed on the console by the initial continuation, the executing agent will find that it no longer has any vertices in its possession whatsoever. At this point, it *could* simply cease to exist, but instead it continues to listen to the network in the hope that some other agent will send it something to do. Thus we can use this trivial case to boot up new agents running on different hosts.

5.2.2 First example

To test the example program, we first run the executable file, passing no command line arguments, on the two hosts we wish to use as servers. This creates two agents (call them A and B) that are simply waiting for messages from other agents. Then we run the executable on a third host, passing it the Agent-Ids of A and B as command line arguments. This creates a third agent (C) who contacts A and B, and together they construct the following initial working graph:

```
(graph ()
  ;; A:
  (<Network Anchor> 10)
  ;; B:
  (<Network Anchor> 9)
  ;; C:
  ;;  Reduction Queue: 3
  (<Global MAIN> 3)
  (<Call 1> target:3 tail:4 cont:8 0:2)
  (<Drop> 4)
  (<Cons> 2 car:10 cdr:1)
  (<Cons> 1 car:9 cdr:0)
  (<Nil> 0)
  (<Network Anchor> 8)
  )
```

(I will be using `;;`-comments to indicate which agents are holding which vertices.) Only one method can be applied to this graph, the one for the connection numbered "3", and both of the vertices that it joins are held by C, so all queues are empty except C's reduction queue. C's scheduler thus calls C's local executive, which applies methods to the subgraph held by C until the working graph becomes:

```
(graph ()
  ;; A:
  (<Network Anchor> 10)
  ;; B:
  (<Network Anchor> 9)
  ;; C:
  ;;  Migration Queue: 10
  (<Call 1> target:10 tail:5 cont:4 0:6)
  (<Lambda 894> 6 n:1)
  (<Number 8> 1)
  (<Evarg 1262> 4 cont:7 args:3)
  (<Cons> 3 car:5 cdr:2)
  (<Cons> 2 car:9 cdr:0)
  (<Nil> 0)
  (<Evarg 1305> 7 cont:8)
  (<Network Anchor> 8)
  )
```

We see that the computation has advanced to the point where A's **`Network Anchor`** is about to be called. The argument being passed is a zero-argument procedure which is a closure with one free variable (whose value is 8). This is the closure from the body of **`REMOTE-FACT`**. The continuation for the call is an

`Evarg 1262` vertex that captures the original `ARGS` argument to `MAIN`. When the call to `FACT` finally returns to the `Evarg 1262`, it will need the value of this variable in order to start evaluating the other argument in the call to `+`.[4] The next continuation down the "stack" is of type `Evarg 1305`, it is waiting to supply the argument to `FORCE-NUMBER`. The bottommost continuation is C's `Network Anchor`, which will display the final value on C's console.[5]

Again only one redex remains, the one corresponding to connection `10`. A still believes that the terminal on the other end of connection `10` is the `car` terminal of a `Cons` (for which there is no method), but C knows the true story, so there is now an entry in C's migration queue. This is the way responsibility for blocked redexes is always handled, the burden always falls on the agent who first discovers the blocked redex. Usually this is due to local execution at that agent replacing a vertex with one of a different type.

C's scheduler sees that nothing more can be done locally, so it allows C's network executive to run, to see if it can unblock connection `10`. The network executive sees that the remote vertex (the `Network Anchor`) is not mobile, but there is nothing to prevent the local vertex (the `Call 1`) from traveling, so it resolves to send a subgraph containing that vertex from C to A.

The network executive now applies its migration heuristics to determine which, if any, other local vertices should accompany the `Call 1` to its destination. These heuristics have yet to be described, but understanding them is not essential to understanding the example. It is sufficient to know that the heuristics decide that it looks like a good bet to pick up *all* the vertices that C is holding (except, of course, C's `Network Anchor`) and send them along to A. Later, in section 5.3, we will see how this advice was computed.

So the network executive calls the subgraph transporter to perform the migration. One message is sent from C to A containing the description of the migrating graph structure, and three link end descriptors:

- one end of the link that carries connection `10`,
- one end of the link that carries connection `9`,
- one end of a new link that carries connection `8`.

After A's subgraph transporter reassembles the structure we have the following situation:

[4] Notice how faithfully this follows the sequential definition of the Scheme language. We don't evaluate any part of the second argument to `+` until after the first argument has actually returned its value.

[5] A network anchor can also be treated as a continuation. The returned value will be printed on the console.

```
(graph ()
  ;; A:
  ;;  Reduction Queue: 10
  (<Network Anchor> 10)
  (<Call 1> target:10 tail:5 cont:4 0:6)
  (<Lambda 894> 6 n:1)
  (<Number 8> 1)
  (<Evarg 1262> 4 cont:7 args:3)
  (<Cons> 3 car:5 cdr:2)
  (<Cons> 2 car:9 cdr:0)
  (<Nil> 0)
  (<Evarg 1305> 7 cont:8)
  ;; B:
  (<Network Anchor> 9)
  ;; C:
  (<Network Anchor> 8)
  )
```

As the subgraph transporter installed the new structure at A it discovered the now unblocked redex corresponding to connection `10`, which is now sitting in A's reduction queue.

When the link layer was informed that one end of the link for connection `9` was moving from C to A, it dispatched an update message from C to B, where it suspected the other end was located. Also, when the link that used to carry connection `10` was destroyed by A, a message was sent to C, the link's home agent. So the entire act of migration involved three messages, plus some acknowledgments.[6]

Now A's local executive gets to work applying methods to the subgraph held by A until the working graph becomes:

[6] The acknowledgment for the structure migration message from C to A can be piggybacked on top of the notice of destruction sent from C to A. The existing reliable message module does not make this optimization, but it should.

```
(graph ()
  ;; A:
  ;;  Migration Queue: 9
  (<Call 1> target:9 tail:0 cont:3 0:1)
  (<Drop> 0)
  (<Lambda 894> 1 n:2)
  (<Number 9> 2)
  (<Evarg 1251> 3 cont:7 0:4)
  (<Number 40320> 4)
  (<Evarg 1305> 7 cont:8)
  ;; B:
  (<Network Anchor> 9)
  ;; C:
  (<Network Anchor> 8)
  )
```

This situation resembles the situation at C just before the previous migration. Again there is a single blocked redex, which is a call on a remote `Network Anchor`. The argument is a closure of the same type, although this time the closed over value is the number 9. The `Evarg 1305` continuation hasn't been touched, but the continuation above it on the stack is now of type `Evarg 1251`. This continuation is holding the value 40320, with the intent of adding it to the returned value.

A similar scene unfolds: The network executive resolves to send the `Call 1` vertex to rendezvous with the `Network Anchor` at B, and the migration heuristics suggest that all of the vertices held by A should accompany it. A message containing a description of the graph structure and two descriptors is dispatched from A to B. The descriptors are for the ends of the links for connections `8` and `9`.

This time the link module sends two updates to C, since it is the home agent for both links, and because it is also the current location of the other end of the link for connection `8`. Also, the link for connection `9` is destroyed at B, generating a message from B to C. Total messages sent for this migration: 4.

The working graph after the migration:

```
(graph ()
  ;; B:
  ;;  Reduction Queue: 9
  (<Network Anchor> 9)
  (<Call 1> target:9 tail:0 cont:3 0:1)
  (<Drop> 0)
  (<Lambda 894> 1 n:2)
  (<Number 9> 2)
  (<Evarg 1251> 3 cont:7 0:4)
  (<Number 40320> 4)
  (<Evarg 1305> 7 cont:8)
  ;; C:
  (<Network Anchor> 8)
  )
```

And after more purely local reduction:

```
(graph ()
  ;; B:
  ;;  Migration Queue: 8
  (<Return 1> target:8 tail:0 0:1)
  (<Drop> 0)
  (<Number 403200> 1)
  ;; C:
  (<Network Anchor> 8)
  )
```

This time instead of calling a remote `Network Anchor`, we are trying to return a value to it, but the effect is much the same: The `Return 1` must migrate from B to C. The migration heuristics again suggest sending the whole show along for the ride. A message is dispatched from B to C containing the graph structure and a descriptor for the link end. This time, the link layer sends no additional messages of its own. Total messages sent for this migration: 1. The result:

```
(graph ()
  ;; C:
  ;;  Reduction Queue: 8
  (<Network Anchor> 8)
  (<Return 1> target:8 tail:0 0:1)
  (<Drop> 0)
  (<Number 403200> 1)
  )
```

which reduces to the empty graph, after printing the number 403200 on C's console.

In all, 8 messages were sent and acknowledged. The current simple-minded reliable message module sends 16 UDP datagrams to transport these messages and acknowledgments. A better reliable message module, which knew how to combine messages and acknowledgments bound for the same destination, could reduce this to 8 datagrams.[7]

More important than counting messages (or the underlying unreliable datagrams) is the total delay introduced by network travel. Three times the entire "task" packed its bags and moved to a new location. Thus the total delay is $3T$, where T is the "typical" network transit time. This is the best possible delay for executing a single task that must visit two remote locations in order to complete its job.[8]

In a system that supported only the standard remote procedure call (RPC) mechanism, where each remote call evokes a reply from the callee back to the caller, the analogous program would run with a delay of $4T$, $2T$ for each of the two remote calls. (Recall that for the moment we are restricting ourselves to purely sequential execution.)

The additional delay of T in an RPC system is an instance of the continuation problem introduced in chapter 1. This example thus demonstrates how my system is able to solve that problem. The crucial moment occurred when the task migrated from A to B. At that time the connection to the continuation on C was passed from A to B along with the rest of the migrating structure. This enabled the task to migrate directly from B to C with the final result. In an RPC system, where there is no explicit representation for a continuation, this would be impossible to accomplish.

Of course the migration heuristics also played a role in this victory over RPC. Things could have gone very badly if the heuristics hadn't made such good choices about what vertices to migrate each time. In section 5.3 we will see why the heuristics are so good at gathering together vertices that correspond to our intuitive notion of a task.

5.2.3 Second example

Now let us make a small change in our example program. Suppose the procedure `REMOTE-FACT` is modified to become:

```
(define (remote-fact host n)
  (future (host (lambda () (fact n)))))
```

Recall that the `FUTURE` special form starts executing its subexpression, but immediately returns to its caller. Thus this new version of `REMOTE-FACT` will return a `Future` to its caller that will eventually *become* the result of calling the `FACT` procedure on the specified host.

[7] The fact that the number of datagrams in the best case works out to be the same as the number of messages is purely a coincidence.

[8] [Par92] takes the trouble to prove this fact!

Everything else about the program and the startup procedure stays the same. The same initial working graph is constructed, and local reduction takes place at C until the working graph becomes:

```
(graph ()
  ;; A:
  (<Network Anchor> 16)
  ;; B:
  (<Network Anchor> 17)
  ;; C:
  ;;  Migration Queue: 16, 17
  (<Call 1> target:16 tail:0 cont:1 0:2)
  (<Drop> 0)
  (<Lambda 1661> 2 n:3)
  (<Number 8> 3)
  (<Future> 13 as cont:1)

  (<Call 1> target:17 tail:4 cont:5 0:6)
  (<Drop> 4)
  (<Lambda 1661> 6 n:7)
  (<Number 9> 7)
  (<Future> 14 as cont:5)

  (<Sum> 15 left:13 right:14)
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

Since futures have been introduced, this initial computation at C is able to advance further before it must involve the network. Both calls to `REMOTE-FACT` returned futures, and the two subgraphs responsible for supplying values for those futures ran until they blocked calling the remote `Network Anchor` vertices located at A and B. Meanwhile, the rest of the graph ran until it got blocked in the code for `FORCE-NUMBER`.

In effect, the working graph forked into three separate tasks (reflected by the grouping of the vertices in the graph expression above), which then ran independently until they blocked. This is what one would expect, given how futures are normally implemented [Mil87, Hal84], but remember that there is no explicit notion of "task" in linear graph reduction. Tasks here are an emergent phenomenon that arises naturally from a more primitive model of computation.

The network executive on C now gets to work on the two blocked redexes: As chance would have it, it first considers connection `17`, whose other end is held by B. It determines that the `Call 1` vertex must migrate to B to rendezvous with the `Network Anchor` there. The migration heuristics are consulted, and they suggest sending the subgraph consisting of the `Call 1` vertex, the directly attached `Drop`, `Future`, and `Lambda 1661` vertices, and the `Number 9` vertex.

These vertices are gathered up and sent in a message to B. A new link is created to support connection `14`. When the graph structure arrives at B, the link for connection `17` is destroyed, generating a message back to that link's home agent, C. Total messages sent for this migration: 2.

Next the network executive considers connection `16`, whose other end is held by A. Again, the `Call 1` vertex must migrate, and this time the heuristics suggest sending attached `Drop`, `Future`, and `Lambda 1661` vertices, the `Number 8` vertex, *and* the `Sum` vertex.

These vertices are gathered up and sent in a message to A. A new link is created to support connection `15`. Since the other end of the link just recently created to support connection `14` is now moving to A, the link layer sends an update to B announcing the move. When the graph structure arrives at A, the link for connection `16` is destroyed, generating a message back to that link's home agent, C. Total messages sent for this migration: 3.

In section 5.3, we will see why the heuristics chose to partition the graph in exactly this manner. For now, simply note that they *almost* selected exactly the "tasks" that we previously identified intuitively. The decision to send the `Sum` vertex to A, instead of keeping it on C, may seem peculiar—in fact, it will cause minor trouble later on—but even that can be defended as a reasonable choice.

After the two migrations the working graph becomes:

```
(graph ()
  ;; A:
  ;;  Reduction Queue: 16
  (<Network Anchor> 16)
  (<Call 1> target:16 tail:0 cont:1 0:2)
  (<Drop> 0)
  (<Lambda 1661> 2 n:3)
  (<Number 8> 3)
  (<Future> 13 as cont:1)
  (<Sum> 15 left:13 right:14)
  ;; B:
  ;;  Reduction Queue: 17
  (<Network Anchor> 17)
  (<Call 1> target:17 tail:4 cont:5 0:6)
  (<Drop> 4)
  (<Lambda 1661> 6 n:7)
  (<Number 9> 7)
  (<Future> 14 as cont:5)
  ;; C:
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

The local executives on A and B now work in parallel applying methods locally until the working graph becomes:

```
(graph ()
  ;; A:
  (<Number 40320> 13)
  (<Sum> 15 left:13 right:14)
  ;; B:
  ;;  Migration Queue: 14
  (<Number 362880> 14)
  ;; C:
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

The network executive on B clearly has no choice but to send the `Number 362880` vertex to A. Since C is the home agent for the link supporting connection 14, this will involve a link layer message from B to C announcing the move, and another from A to C when the link is destroyed. Total messages sent for this migration: 3. The result:

```
(graph ()
  ;; A:
  ;;  Reduction Queue: 14
  (<Number 40320> 13)
  (<Number 362880> 14)
  (<Sum> 15 left:13 right:14)
  ;; C:
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

A's local executive reduces this to:

```
(graph ()
  ;; A:
  ;;  Migration Queue: 15
  (<Number 403200> 15)
  ;; C:
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

A's network executive then sends the `Number 403200` back to C. Since the home agent for the link supporting connection 15 is also C, no link layer messages are needed. Total messages sent for this migration: 1. The result:

```
(graph ()
  ;; C:
  ;;  Reduction Queue: 15
  (<Number 403200> 15)
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

which reduces to nothing, after printing the number 403200 on C's console.

In all, 9 messages were sent and acknowledged. The current simple-minded reliable message module sends 18 UDP datagrams to transport these messages and acknowledgments, but as before, a better reliable message module could reduce this to 8 datagrams.

The network induced delay is again $3T$, because of the critical path from C to B to A and back to C. This is where the decision to migrate the `Sum` vertex gets us in trouble. If the `Sum` had remained behind on C, then when the computations on A and B completed they would have both sent their results directly to C, where they would have been added and immediately printed. In this case the delay would have been only $2T$.

In a standard RPC system that also supported some kind of futures (such as those described in [LS88]) the analogous program would in fact run with a delay of $2T$—so in this case RPC wins. The migration heuristics had to guess about the best place to compute the sum (which is computed after a delay of $2T$ in either case), and they got it wrong. RPC is too inflexible to even consider the possibility of computing the sum elsewhere, and so in this case it happens to do the right thing.

I could have fiddled with the heuristics to make the optimal thing happen in this example as well, but the goal of this second example was to illustrate the effect of introducing futures, not to beat RPC a second time. It also helps to have an example of the heuristics performing less than perfectly in order to emphasize that they are, after all, only heuristics.

5.3 Migration heuristics

Until now I have avoided explaining the heuristics employed by the network executive to select migratory subgraphs. I did this to emphasize how these heuristics are not in any way essential to the system as described so far. As long as the network executive migrates the vertex necessary to unblock a blocked redex, the heuristics can do very little to prevent the computation from making at least *some* progress. (That is, as long as they don't create *new* blocked redexes—fortunately this is easy to forbid.)

Of course *good* heuristics can improve matters a great deal by anticipating where a vertex is going to be needed, and sending it there in advance. In the examples of the last section it was clearly beneficial for the arguments attached to a migrating `Call` to accompany the `Call` to its destination. But I make no claim that the heuristics described here are the *right* set of heuristics; I've only experimented with a few variations—the current set work acceptably well—but much better heuristics are clearly possible.

These heuristics are interesting, not because of exactly what they do, but because of *the kind of reasoning that went into creating them.* The key idea is to take advantage of the fact that linear connections behave in a more predictable fashion than nonlinear reference mechanisms.

These heuristics examine graph structure and make judgments about the consequences of moving bits of it from agent to agent. They have no understanding of what the vertex types meant to the compiler (as procedure calls, closures, continuations, or whatever), all they know about the vertices is the terminal activity information (described in chapter 3). They compare different distributed configurations of graph structure by comparing the links required to support each configuration, and reasoning about the expense likely to result from each link.

The costs associated with each link are easy to estimate in part because of the pains we took in chapter 4 to limit the expense of each link level operation (creating, moving and destroying them). More importantly, link costs are easy to estimate because a typical link is used exactly *once*. An agent will use a link to track down the location of some remote vertex, and will then send the local vertex it holds to that location. When the migrating vertex arrives, the link will be destroyed, as it is no longer needed. (This was the fate of every single link in the two examples in the last section.) To a first approximation every link represents exactly one future migration, so a strategy for minimizing the number of migrations is to try to minimize the number of links.

Another way to understand why this works is to think of a link as representing a *capability* to send *one* message between a pair of mobile entities. Limiting the capability to a single message enables us to reclaim the resources devoted to maintaining the link at the same time the message is sent; this keeps costs incurred by a link during its lifetime fixed. If multiple messages are required, multiple links must be forged in advance of those requirements. The result is that the system can anticipate to some extent how many messages the current computation will need to exchange, at least in the near term, by counting the number of extant links.

There is a good analogy between the way blocked redexes are handled and the way page faults are handled by demand paging. In both cases, the possibility of the fault is ignored at compile-time, and instead the fault is detected and handled at run-time. In demand paging, the fault is detected by the memory hardware. In distributed graph reduction, the blocked redex is detected when `connect_dispatch` is passed a value tagged as a connection to a remote terminal. In demand paging, adjacent words are read into physical memory in addition to the single word that is required to satisfy the fault; this simple heuristic takes advantage of locality of reference to decreases the number of future page faults. In distributed graph reduction, the heuristics described in this section play a similar role; they work to decrease the number of future faults by anticipating redexes and transporting additional graph structure. In both cases the run-time can afford to do a certain amount of head-scratching, because it is already committed to an expensive action to clear up the existing problem.

5.3.1 The heuristics

Once an agent's network executive has decided to migrate a single vertex it works on expanding the migratory subgraph one vertex at a time. Each locally held vertex that is directly connect to a vertex that will migrate, is considered in turn. If that vertex satisfies the criteria described below, it is added to the migratory subgraph. This process of examining the fringe of the migratory subgraph is repeated until eventually no vertices satisfy the criteria.

Since this algorithm only considers the effects of adding one vertex at a time, it performs a rather myopic hill-climb search for the best subgraph to migrate. It is not hard to construct cases where adding either of two vertices alone will be rejected, but adding both both vertices at once would be an improvement. This algorithm will never discover such possibilities.

Don't let the message get too large. As the migratory subgraph grows, the size of the message that will be sent generally grows. Each vertex to be migrated takes a certain amount of space to describe, and each link end descriptor takes up space. But vertices are typically somewhat smaller than descriptors,[9] so if sending a particular vertex will decrease the number of required links, then the message size will actually decrease. Thus, if the algorithm reaches a point where the message is close to the maximum message size, it will only consider additions to the subgraph that actually decrease the number of links.

Don't migrate unblocked redexes. If a vertex is a member of an unblocked redex, it will never be considered for migration. This prevents agents from exporting unfinished work to other agents—the idea is to let computation finish running locally before getting involved with the relatively expensive network. As a special case, this restriction prevents migration from creating new blocked redexes.

Monoactive vertices follow their active terminal. Vertices that have only one active terminal and vertices that have many active terminals are treated differently. The majority of vertices fall into the monoactive case. Such a vertex is added to the migratory subgraph if and only if its active terminal is connected to a vertex already selected for migration.

A monoactive vertex cannot participate in any future reductions unless its sole active terminal gets connected to some appropriate vertex (one for which a method exists). If a monoactive vertex gets separated from whatever graph structure its active terminal is connected to, nothing further can happen to it unless it subsequently packs up and follows that structure, or unless that structure happens to return to it. So letting such a separation take place *guarantees*

[9] In the current system, on the average, three vertices take up the same amount of space as a single descriptor.

that a message must be sent before the monoactive vertex can do anything useful. If we encourage the monoactive vertex to follow what its active terminal is connected to, many such messages will be avoided. When that structure finally becomes something the vertex can interact with, the vertex will already be there ready to participate in a local reduction.

As an important special case of this heuristic, a vertex with only a single terminal will always follow whatever structure it is attached to. Thus numbers and other atoms will always stick with the vertices they are connected to.

This heuristic is largely responsible for the way the system so successfully extracts "tasks" from the working graph. Continuations are represented using vertices whose sole active terminal is connected to the graph structure that will eventually return a value. So continuations will tend to tag along after the computations whose values they are waiting for. Continuations linked together to form a "stack" will tend to travel together.

An essential ingredient in the way stacks stick together is the fact that continuations are almost always treated linearly in Scheme programs. (The only exception is when `CALL-WITH-CURRENT-CONTINUATION` is used. See section 3.1.3.1.) Due to this linearity, each continuation will be directly connected to the next, without even an intervening tree of `Copy` vertices. This is a clear example where the linearity in the original program, exposed by the explicit representation as linear graph structure, is exploited to aid in the execution of the program.

For polyactive vertices, just avoid creating more links. Polyactive vertices are added to the migratory subgraph as long as doing so does not *increase* the number of links. The idea here is that as long as no additional links are required, a vertex is more likely to be needed at the remote agent, where we know some action is about to take place, than it is locally, where things have settled down.

Combining all the above heuristics yields the following algorithm:

Step 0 Initialize G to contain just the initial vertex that is to be migrated.

Step 1 For each local vertex $v \notin G$, where v is not a member of an unblocked redex, and v is connected to some vertex $w \in G$ do the following:

- If sending v would strictly *decrease* the number of required links, add v to G.
- If the message being assembled is not close to full, and sending v would not *increase* the number of required links, add v to G.
- If the message being assembled is not close to full, and the terminal through which v is joined to w is monoactive, add v to G.

Step 2 If any new vertices were added to G in step 1, go do step 1 again.

5.3.2 The example revisited

Let us now return to the following situation from the previous section, and examine in detail exactly how the two migratory subgraphs were chosen:

```
(graph ()
  ;; A:
  (<Network Anchor> 16)
  ;; B:
  (<Network Anchor> 17)
  ;; C:
  ;;  Migration Queue: 16, 17
  (<Call 1> target:16 tail:0 cont:1 0:2)
  (<Drop> 0)
  (<Lambda 1661> 2 n:3)
  (<Number 8> 3)
  (<Future> 13 as cont:1)

  (<Call 1> target:17 tail:4 cont:5 0:6)
  (<Drop> 4)
  (<Lambda 1661> 6 n:7)
  (<Number 9> 7)
  (<Future> 14 as cont:5)

  (<Sum> 15 left:13 right:14)
  (<Copy> target:15 a:8 b:9)
  (<Equal?> 11 left:8 right:10)
  (<Number 0> 10)
  (<Test 746> 11 cont:12 n:9)
  (<Network Anchor> 12)
  )
```

Recall that the network executive started by considering connection `17`, and determined that it should send the `Call 1` at one end of that connection from C to B. Here is how the migration heuristics work in that case:

1. The `Drop` connected to the `tail` terminal of the `Call 1` is sent because its sole active terminal is connected to the `Call 1`.

2. The `Lambda 1661` connected to the `0` terminal of the `Call 1` is sent for the same reason.

3. The `Number 9` connected to the `n` terminal of the `Lambda 1661` is sent to follow the `Lambda 1661`.

4. The `Future` connected to the `cont` terminal of the `Call 1` is sent because if it isn't sent, connection `1` will require a link, while if it is sent, connection

13 will require a link. So sending it creates no increase in the number of links.

5. The Sum vertex is *not* sent. If it isn't sent, connection 13 will require a link, while if it is sent, connections 14 and 15 will both require links.

As the Sum vertex was the only remaining candidate for migration, at this point the migratory subgraph stops expanding.

When the network executive considers connection 16 and its Call 1 vertex, everything happens in much the same way until the Sum vertex comes up for consideration. This time, if the Sum is not sent, connections 13 and 14 will require links. (Remember we made a link for connection 13 because of the previous migration!) If the Sum is sent, connections 13 and 15 will require links. Either way, we need two links, so the Sum gets migrated.

The next step is to consider the Copy vertex. If it is not sent, connection 15 continues to need a link, but if it is sent, connections 8 and 9 both require links. Thus the Copy is not sent, and the migratory subgraph stops expanding.

It would not be hard to change these heuristics so that in this example the Sum vertex was *not* migrated; we could insist that in the polyactive case vertices are only migrated if they actually decrease the number of required links. This would cause the delay in printing the answer to drop from $3T$ to $2T$. But remember that the interesting thing about these heuristics is not how they function, but the reasoning that went into creating them. Rather than patching them so that particular cases function in particular ways, we should examine this example to see if it reveals some flaw in the reasoning behind the heuristic.

Note that while the delay before the answer is *printed* is $3T$, the delay before the answer is *computed* is $2T$. In fact, the delay will be $2T$ whether the Sum vertex is migrated to A, or remains on C. So the real problem with sending the Sum to A is that it causes the answer to the call to + to appear in the wrong place, given that the rest of the computation calls for that value to be printed by C.

Could *any* heuristic predict this, given the state of the working graph at the time the migration of the Sum is being considered? The heuristic must choose between two nearly symmetrical situations. In either case, the Sum vertex winds up on one agent, with one connection to some local structure, and two connections to remote structures. In order to make an informed decision about which alternative to select, the heuristic needs to understand more about the future course of the computation than can be extracted just by counting links. It is possible to construct cases where we are given a similar working graph, but due to the future course of the computation, the best choice *is* to migrate the vertex. Simply changing the comparison in the heuristic from $\leq$ to $<$ doesn't actually address the problem.

In order to distinguish between the two cases in the Sum example, the heuristics need to know more about the universe of methods—they need to know more about the consequences of their actions. The terminal activities are one step in that direction, and we have already seen that the heuristic for monoactive vertices uses that information to good advantage. Perhaps some additional information precomputed from the universe of methods can help suggest that it is not really a good idea for a `Sum` vertex to get separated from the structure attached to its handle.

The preceding discussion should not in any way belittle the performance of the existing heuristics. They work quite well at exploiting the existing combination of linearity and terminal activity information. I'm only suggesting that by continuing to travel in the same direction, the system can be made even better.

5.4 Summary

This completes the presentation of the practical implementation of distributed linear graph reduction. In chapter 3 I showed how to translate an ordinary sequential Scheme program into a linear graph grammar. A key property of this translation is that linearities in the original program remain linearities in the resulting grammar. Chapter 4 demonstrated that linear references between distributed entities can be maintained cheaply. The fact that linear references are more constrained that full-fledged pointers makes this protocol simple, cheap and fast. In this chapter, the pieces were put together to build a system that executes the translated program in a distributed environment. Linear references also helped here by allowing the run-time system to make good decisions about how to distribute the graph structure.

This system is quite real. The run-time system contains 5400 lines of C code. It runs under several popular versions of Unix. The compiler consists of 2500 lines of Scheme code. It tries to honestly address all of the issues that would arise in a "production" system. This includes issues, many of them quite mundane, that I have not even mentioned here. For example, there are interesting techniques used for describing graph structure compactly in messages, and for representing graph structure so as to make local reduction run faster.

Many things have been left undone. The rest of this section catalogs some of the issues that would have to be addressed before this system could be used for real.

Fault Tolerance. The current system is not at all fault tolerant. If an agent crashes, the system breaks completely at all levels from the reliable message layer on up. In a production system, something would have to be done about graph structure that was temporarily inaccessible due to problems with an agent or the network path to that agent. For example, graph structure en route to such an agent could be returned to the sender.

Perhaps the same kind of reasoning about linear graph structure that went into the design of the migration heuristics could help in coping with faults. This might be a fruitful area of future research.

Security. Nothing in the current system addresses security concerns. The current run-time is even perfectly happy to migrate the company payroll database (represented as linear graph structure) to the workstation sitting on the receptionist's desk.

Better Link Protocol. As noted at the end of chapter 4, there are two major improvements to be made to the link maintenance protocol. First, the link layer should be able to relocate the home agent when neither end of the link remains at home. Second, the link layer should be more closely integrated into the reliable message layer.

Migration. As noted at the end of the previous section, more intelligent migration strategies are clearly possible, and probably desirable. Other factors besides blocked redexes and potential links can be taken into consideration. Subgraphs should be migrated from overloaded agents to underutilized agents. The cost of a link should be weighted according to the estimated network delay between its two ends. These strategies, and others like them, would continue to build on the same basic foundation that supports the existing migration strategies.

Thrashing. Nothing prevents migrating graph structure from chasing other migrating graph structure uselessly around the network. In the simplest case of this problem two agents holding vertices on opposite ends of a connection simultaneously decide to send the vertex they are holding to the other agent. The result is that the two vertices change places, and nothing useful has been accomplished.

Some simple strategies for preventing such "thrashing" are already built in to the current migration heuristics (this is one reason why the agent who first discovers a blocked redex is always responsible for unblocking it), but in a production system some additional anti-thrashing mechanisms might well be needed. It is possible that random variations in the behavior of the network will be sufficient to prevent graph structure from chasing its tail forever, but if reliance of natural randomness proves insufficient, some simple tie-breaking strategies can be implemented.

This is an example of the kind of issue that can't really be addressed without first gaining some experience with running large applications.

Dynamic Code Distribution. Currently all methods are known by all agents before the system starts running. This makes it impossible to introduce new vertex types after the program has started running. Unfortunately, this is precisely what an agent that offers some general service to the network needs to be able to do. Such agents cannot possibly come into existence already knowing all the programs that they might need to briefly host. Instead when they see a vertex type that have never encountered before, they need to go and learn the methods they may need to handle that type.

This requires some straightforward work on the run-time to cause it to notice when a method might be missing and search for it. Presumably such a "method fault" is a rare occurrence, since the results of the search can be cached locally. The resulting algorithm should closely resemble demand paging.

Nonlinear References. There are some situations in which linear references perform poorly, so a true production system would also include nonlinear reference mechanisms. For example, an object that represents the root of a file system might have hundreds of outstanding references. In this case, building a tree of `Copy` vertices to collect and serialize operations might be very inefficient—especially in a distributed implementation where operations might have to travel over the network from agent to agent tracing connections between widely scattered `Copy` vertices.

A better mechanism might be a true "pointer" data type. Clients would be able to freely transport and duplicate copies of a pointer, but the target of a pointer would remain ignorant of how many copies currently existed and where they were located. The target would therefore find it much more difficult to move to a new location, or to terminate its existence. Pointers would be an alternative to connections that would provide a different, and sometimes more appropriate, allocation of the responsibilities and expenses associated with keeping a reference to an object.

A good analogy can be made between the "pure" linear graph reduction system I have presented and an implementation of "pure" Lisp. A pure Lisp implementation might only support a `Cons` data type, but a true production version of Lisp supports data structures such as arrays that have efficient implementations in terms of the underlying hardware.

Garbage Collection. Methods for handling the interaction of vertices with `Copy` and `Drop` vertices gives the system the equivalent of a reference-count garbage collector. As with any reference count garbage collector, circular structures can evade the reference count mechanism.

Full Scheme. As has already been mentioned, the run-time only supports a limited subset of the features of a full Scheme implementation. Much of the work required to complete this job is straightforward and dull, but there are interesting language design issues still waiting to be addressed. Designing an I/O system is one unfinished job we have already encountered.

Another highly interesting problem is supporting the Scheme procedure `EQ?`. The current protocol for representing objects does not include any way to test if two references actually speak to the "same" object. In other words, there is no way to tell if two connections are connected to the fringe of the same `Copy` vertex tree.

Any treatment of this issue must address subtle issues about exactly what one means when one talks about "the same object". These issues are the subject of chapter 6.

Language Design. Finally, it is worth mentioning that a production system built around distributed linear graph reduction would not necessarily want to stick so closely to a sequential programming language semantics. This demonstration system uses standard sequential Scheme in part to demonstrate that it could be done, and in part to avoid overloading the reader with too many new concepts.

A production system would at least introduce some new programming language extensions that help promote linearity and that allow parallelism (such as the `FUTURE` special form). But a new programming language could also be designed from scratch, perhaps by using a different protocol for continuations.

After presenting a catalog of things left undone and paths still to be explored, let me finish the chapter in a more positive way with a summary of what the distributed linear graph reduction engine has demonstrated about the benefits of paying attention to linearity and of using linear graph structure:

- All run-time structures (continuations, record structures, procedures, etc.) are represented explicitly using linear graph structure, so the proper treatment of continuations in tail-recursive procedure calls is ensured.
- Linearity keeps the link protocol simple, so cross-network references are cheap.
- Cheap cross-network references permit data structures to be highly portable. ∎
- Linearity makes it possible to forecast future network traffic, so heuristics can be designed that facilitate the demand migration of tasks and data.

Combine these benefits and the result is that programs such as the first example in section 5.2 can function with the *minimum possible* network delay.

Chapter 6

State

Having demonstrated how linear graph reduction functions in a practical setting in the last three chapters, I would now like to turn to a more theoretical application. In this chapter the notion of linearity and the tool of linear graph reduction will be used to examine one of the most perplexing phenomenon in computer science: the phenomenon of *state*.

As functional programming languages and parallel computing hardware become more widespread, understanding the phenomenon of state is becoming increasingly more important. 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 [Bac78, Agh86, GL86, Kni86].

I believe that before either outlawing state, or learning to simply tolerate it, we should try to better understand it, in the hope of eventually being able to reform it. This chapter takes some steps towards such an understanding.

Using the linear graph reduction model we will be able to characterize those systems in which some components of a system perceive other components as having state. We will learn a new way of thinking about state, and we will gain insight into why state seems to be such a problem. This insight might one day help us make our programming languages more expressive when we program with state.

This excursion into the theory of state may seem quite abstract, especially in contrast to the practical presentation of the distributed graph reduction system just concluded. The two may seem quite unrelated, but there are at least three important points of contact between them. First, the problem of managing state in a distributed environment is quite an important one. A distributed database is chiefly a system for providing the useful facility of state to a distributed set of clients. Since linear graph reduction provides insight into both state and distributed computing separately, it may prove profitable to apply these ideas in fields that have both aspects, such as distributed databases. I will not, however, be demonstrating such a combined application here. This remains a promising research topic.

The second point of contact is of a more practical nature. Having seen linear graph reduction applied in a real system will make much of the following presentation much easier to understand. The subtleties of how linear graph structure can be used to reproduce the behavior of more familiar computational systems have already been explained in more than enough detail. I will be able to assume that the reader has gained a certain amount of intuition about how linear graph reduction works in practice.

Third, the contrast between the two applications, one quite practical and applied, and the other quite theoretical and abstract, serves to underscore my contention that linearity is an important notion of quite general applicability that deserves to be more widely appreciated throughout computer science.

6.1 What is state?

It is not immediately clear to what, if anything, the word "state" refers. 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, as I will argue below, 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 [SS78] 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 [Hen80] 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 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 [DG73], 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.

6.2 The symptoms of state

Experience using linear graph reduction suggests that all graph 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 linear 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 already has an adequate definition. 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.

6.2.1 Symptom: nondeterminism

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

Recall once again the protocol for messages, first introduced in section 2.4. A pair of methods, such as the two on page 28, permit a message (such as `Car`) to climb up through a tree of `Copy` vertices, using two terminals customarily labeled `target` and `tail`. Such methods are a source of nondeterminism because in a graph such as

```
(graph (0 3 4 5 6 7)
  (<Copy> target:0 a:1 b:2)
  (<Car> target:1 tail:3 cont:4)
  (<Set Car> target:2 tail:5 cont:6 new:7))
```

there is a choice about which message to propagate through the `Copy` vertex first. The result might be either

```
(graph (0 3 4 5 6 7)
  (<Car> target:0 tail:1 cont:4)
  (<Copy> target:1 a:3 b:2)
  (<Set Car> target:2 tail:5 cont:6 new:7))
```

or

```
(graph (0 3 4 5 6 7)
  (<Set Car> target:0 tail:2 cont:6 new:7)
  (<Copy> target:2 a:1 b:5)
  (<Car> target:1 tail:3 cont:4))
```

Depending on this choice, either the `Car` message, or the `Set Car` message will arrive at the apex of the tree first—two completely different computational histories may then unfold. This nondeterminism is possible because `Copy` vertices are willing to interact with vertices connected to either their `a` or `b` terminals. Intuitively, a `Copy` tree is willing to "listen" to messages arriving from anywhere along its fringe.

So nondeterminism is built in to the usual linear graph reduction implementation of objects with state. Of course this does not constitute proof that *any* implementation of objects with state must contain nondeterminism. Lacking a definition of state (constructing such a definition is the ultimate goal of this chapter) no such proof is possible.

Still, it is hard to imagine how this nondeterminism can be eliminated. Consider what would happen if `Copy` vertices were only willing to interact through their `a` terminals. In that case there would only be a single location on the fringe of a `Copy` tree that would 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 *in advance* the order in which reads and writes will take place. It would be impossible to support patterns of reads and writes that varied dynamically. It is clear that such a system would fall short of supporting what most programmers would consider state-like behavior.

Instead of simply altering the protocol for `Copy` trees, perhaps some completely different technique for translating a programming language can be found. A technique that builds some other form of graph structure when a name is used more than once, and such that a deterministic grammar can work on that structure to produce apparent state-like behavior. But no such technique is known; nondeterminism appears in every fully satisfactory linear graph reduction implementation of state-like behavior.

Others have also observed that the need for nondeterminism seems to be a symptom of the desire for state. In [Hen80], 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.

6.2.2 Symptom: cycles

Why should it be the case that cyclic linear 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 linear graph external to him behaves as if it had state.

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

```
(define (experiment x)
  (begin (set-car! x 1)
         (= (car x) 1)))
```

The general idea is to detect that the external system, accessed through the variable `X`, somehow *remembers* the action performed by `SET-CAR!`, and this can be detected by the procedure `CAR`. The programmer who wrote this procedure probably thought in terms of some object (probably a `Cons`), named by the variable `X`, whose state could be written and read by `SET-CAR!` and `CAR`.

The important thing to notice about the procedure `EXPERIMENT` is that the variable `X` occurs in its body twice. The reason for this is that two references to the subsystem being tested are needed in order to complete the experiment. While one reference is passed to `SET-CAR!`, a second reference must be retained so that `SET-CAR!`'s effects can be observed. When `EXPERIMENT` is translated in linear graph reduction methods (using the techniques from chapter 3), the first method is the following:

```
(graph (0 1 2)
  (<Global EXPERIMENT> 3)
  (<Call 1> target:3 tail:0 cont:1 0:2))
(graph (0 1 2)
  (<Global EXPERIMENT> 0)
  (<Evseq 401> 3 cont:1 x:4)
  (<Number 1> 5)
  (<Set Car> target:2 tail:4 cont:3 new:5))
```

A picture of this method appears in figure 6.1. Notice that *two* connections join the `Evseq 401` vertex to the `Set Car` vertex. The first connection is through the `cont` terminal of the `Set Car`, because the `Evseq 401` is a continuation waiting for confirmation that the call to `SET-CAR!` has completed. The second connection joins the `tail` of the `Set Car` to the `x` of the `Evseq 401`, because after that confirmation arrives, the continuation will need that second copy of the reference to the value of `X` to continue the experiment.

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

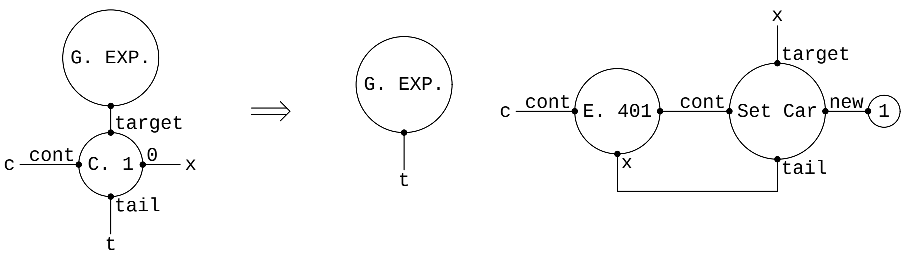

Figure 6.1: The method for calling **EXPERIMENT**

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 linear graph structure 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 linear 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 [SS78] 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 without 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.

6.3 Locality

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

In this section the simplicity of the linear 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 linear 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 (two vertices and a single connection). Thus, there is no way for a running program to learn anything about the nonlocal structure of the linear 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, how message passing and procedure calling were 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 configured as expected. Recognizing this need to *trust* in the global configuration of the system will be the key to a new understanding of state.

6.3.1 Homomorphisms and local indistinguishability

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

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

ψ is an **epimorphism** if it is onto, a **monomorphism** if it is one-to-one, and an **isomorphism** if it is both. If ψ is an isomorphism, it has an inverse ψ^{-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 $\psi(v)$ be the corresponding vertex in H. In fact, a homomorphism is completely determined by its action on vertices.

Figure 6.2 shows an example of a homomorphism.[1] 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 linear graphs, although in general there may be many.[2]

Imagine what it would be like to explore a maze that was built on the plan of a linear 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 before. For all the explorer can tell, the linear 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 linear graphs in figure 6.2. Such graphs are locally indistinguishable.

Formally, H_1 and H_2 are **locally indistinguishable**, written $H_1 \sim H_2$, if there exists a graph G and two epimorphisms $\psi_1: G \to H_1$ and $\psi_2: G \to H_2$. It can be shown that local indistinguishability is an equivalence relation on linear graphs. As a special case of this definition note that if $\psi: G \to H$ is any epimorphism, then $G \sim H$.

---

[1] This picture resembles a picture of a method since it has a left hand side and a right hand side and arrows that express a relationship between the two. This resemblance is coincidental—the two notions will be kept entirely separate.

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

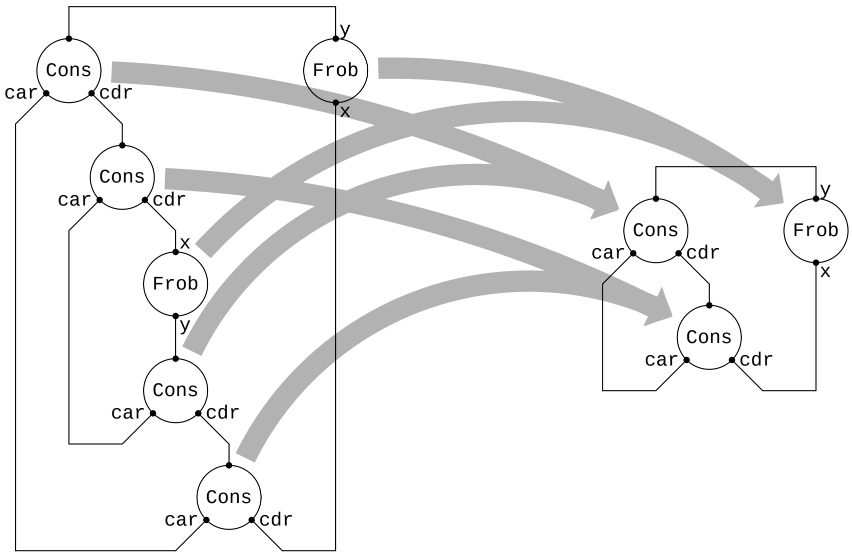

Figure 6.2: A Linear Graph Homomorphism

6.3.2 Methods

Things become more complicated once we introduce methods into the picture. In this section we will prove some theorems about the relationship between linear 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 will continue to assume that all methods are binary methods. The theorems in this section are all true even if we slightly relax that restriction and allow any method whose left hand side is a tree (a connected graph containing no cycles), but that additional generality is not needed in anything that follows.

We write $G \Rightarrow G'$ if the linear graph G' is the result of applying any number of methods to any number of *disjoint* redexes in 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 $\psi: G \rightarrow H$, and if $H \Rightarrow H'$, then there exists a linear graph G' and a homomorphism $\psi': G' \rightarrow H'$ such that $G \Rightarrow G'$. This can be summarized in the following diagram:*

$$\begin{array}{ccc} G & \xrightarrow{\psi} & H \\ \Downarrow & & \Downarrow \\ G' & \xrightarrow{\psi'} & H' \end{array}$$

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

Proof. Each redex in H that is reduced in forming H' can be lifted back through ψ to a set of redexes in G. The set of all such redexes can then be reduced to obtain G'. ψ' can then be constructed from ψ in the obvious way. □

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

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

The theorem 2 is true given any linear graph grammar. It is the strongest such result I have proven 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 linear graph grammar is **preclusive** if two redexes can never overlap. This means that if a redex appears in a linear graph G, and if $G \Rightarrow G'$, and if that redex was not one of the ones reduced in forming G', then that redex still appears in G'. The appearance of a redex in a graph thus *precludes* the possibility that anything else will happen to those vertices before the corresponding method can be applied.

For example, any grammar that contains the two methods on page 28 cannot be preclusive. The reason for this is that, as we showed in section 6.2.1, it is possible to construct a graph where the `Copy` vertex belongs to two different redexes. We have already identified this property of `Copy` vertices as a source of nondeterminism. The following theorem demonstrates that preclusive grammars are in fact deterministic.

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

Proof. Since the grammar is preclusive the redexes in G are all disjoint. We can divide them up into four classes, (1) those that were reduced in forming both G_1 *and* G_2, (2) those that were reduced *only* in forming G_1, (3) those that were reduced *only* in forming G_2, and (4) those that were reduced in *neither* case. Redexes in the second class must still occur in G_2, and redexes 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 redexes.) □

Theorem 4 *If a linear graph grammar is preclusive, then given linear graphs G, G_1, and G_2 such that $G \Rightarrow^* G_1$ and $G \Rightarrow^* G_2$, 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 linear graphs. It shows that no matter what order we choose to apply the methods from a preclusive grammar, we always achieve the same result. If it is possible to apply methods until a linear graph is produced to which no further methods can be applied, then that graph is unique.

The final two theorems relate local indistinguishability and preclusive grammars:

Theorem 5 *If a linear graph grammar is preclusive, then given linear graphs* G*,* H*, and* H' *such that* $G \sim H$ *and* $H \Rightarrow H'$*, there exists linear 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 reducing *all* redexes in G and H. This is possible because these redexes must all be disjoint (since the grammar is preclusive). Further, it must be the case that H' is the result of performing some subset of these reductions, 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 linear graph grammar is preclusive, then given linear graphs* G*,* H*,* G'*, and* H' *such that* $G \sim H$*,* $G \Rightarrow^* G'$ *and* $H \Rightarrow^* H'$*, there exists linear graphs* G'' *and* H'' *such that* $G' \Rightarrow^* G''$*,* $H' \Rightarrow^* H''$ *and* $G'' \sim H''$*. This can be summarized in the following diagram:*

$$\begin{array}{ccc} G & \sim & H \\ \Downarrow * & & \Downarrow * \\ G' & & H' \\ \Downarrow * & & \Downarrow * \\ G'' & \sim & H'' \end{array}$$

Proof. This follows from the previous theorem by induction and by using theorem 3. □

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

6.4 Implications for programs

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

Suppose we are given a pair of linear graphs G and H, where $G \neq H$, and we are asked to produce a linear 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 $\psi: G \to H$. 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 $\psi': G' \to H'$. Since ψ' 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. Thus no grammar can ever *correctly* decide 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 $\psi: G \to H$, 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. Everything that can happen to H is locally indistinguishable from something that can also happen to G.

Looking at this fact from yet another angle: If a system is halted in configuration H, and reconfigured to be in configuration G, where there is an epimorphism $\psi: G \to H$, the system can perhaps "malfunction" by arriving at a configuration G' (i.e. $G \Rightarrow^* G'$) where there are no configurations G'' and H'' such that $G' \Rightarrow^* G''$, $H \Rightarrow^* H''$ and $G'' \sim H''$.

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 $\psi: G \to H$ 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. Surprisingly, these are almost exactly the two symptoms we previously identified as always being present in systems that exhibit state-like behavior. (I say "almost exactly" because I never precisely defined what was meant by "nondeterminism" in section 6.2.1. The content of theorem 4 was that preclusive grammars behave deterministically, but there are non-preclusive grammars that also behave deterministically, so the two concepts do not align exactly.)

We have now arrived at the crucial intersection of our intuitive observations about state-like behavior (section 6.2) with our theorems about locality (section 6.3). We have discovered that the two features that always accompany state like behavior are just the features necessary to make the system dependent on its nonlocal structure. 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.

Accepting this proposed definition of what it means for a system to experience state, leaves us with the following picture of the world: Stateless systems have the property that they are insensitive to their nonlocal structure—they behave the same way in all locally indistinguishable configurations. State is experienced by the components in a system when some locally indistinguishable configurations of the system may evolve in additional unintended ways. Importantly, no test the system can perform internally can determine that it is properly configured.

This says a great deal about why programming in the presence of state is difficult. Programming with state means that there are conditions which the system depends upon, that it cannot check for itself. The system must *trust* that it is configured as it expects. It must trust that it evolved from known initial conditions through the application of known methods, rather than being created spontaneously in some locally indistinguishable configuration that could never have been reached naturally.

6.4.1 The parable of the robot

To make this more concrete, recall the robot from section 6.1 who was interacting with his 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 configuration. Specifically, replace it with 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 configuration, each robot and each computer must be placed in the same internal configuration as it was when 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.

6.5 The object metaphor

We have concluded 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 job.

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 [GR83, LAB$^+$81, Moo86, Agh86] 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.

When we started our investigation of state we adopted the view that programming languages are designed for instructing the entities that inhabit the computational universes we create. These embedded observers are the ones that perceive state in the other components of the universe that surrounds them.

Given our identification of state-like behavior as dependence on the nonlocal structure of the system as a whole, there is no apparent reason to suppose that the object metaphor is the *only* way to describe state-like behavior to those embedded observers. The word “object” does not appear anywhere in section 6.4, so perhaps we can discover new metaphors that capture the way state “feels” to an embedded observer, without reference to the notion of object.

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.

6.6 Summary

In this chapter the notion of linearity and the tool of linear graph reduction was used to examine the phenomenon of state. We concluded that state occurs when some locally indistinguishable configurations of the system may evolve in unintended ways, and furthermore no test the system can perform internally can determine that it is properly configured. This explains why programming in the presence of state is difficult—such systems must trust in untestable properties. My hope is that this insight will someday lead to an improvement in the terminology used to describe state in our programming languages.

As mentioned in the introduction, I also hope that someday this insight can be profitably combined with the more practical application of linear graph reduction to distributed computing.

Finally, let me remark once again on the striking contrast between the two applications of linearity presented herein. This chapter used linear graph reduction to achieve a deep theoretical insight, while the previous three chapters used the very same tool to address some highly practical problems. Linearity is a very flexible tool.

Chapter 7

Conclusion

This chapter puts linear graph reduction in context by (1) comparing it to related and similar work on naming, programming languages, distributed computing, graph reduction and state, and (2) describing some of the future research areas that can build on what I have done.

7.1 Relation to other work

The importance of linearity in naming has never been explicitly highlighted as I have done here, but the phenomenon has always existed. Once sensitized to the issue of linearity, you can find instances of it throughout computer science. Some examples are described below.

Variants on linear graph reduction itself have also appeared. Given the simplicity of the basic model, this isn't surprising. I like to imagine that linear graph reduction occupies a point of minimum complexity in some space of computational models—anyone searching for a simple linear computational model is likely to be drawn towards it once they get close enough.

Distributed and parallel computing are hot topics these days, so it's not surprising that there are many connections between the system described here and much recent work in that area. In contrast, my work on state appears to be almost unique.

7.1.1 Linear naming

There are some programming languages that have adopted a linear restriction on the way that variables can be used. Janus [SKL90] is a concurrent logic programming language where only two occurrences of any given variable are allowed in any clause. When one occurrence is used to constrain the value of the variable, the other occurrence can be used to read that value. This is the logic programming equivalent of the notion of a linear name. The authors adopt this restriction because it simplifies the semantics of their language so that they can regard variables as point-to-point communication channels. They sometimes call this property being "broadcast-free", a phrase that clearly points to the linear origins of the restriction.

In [Laf90] Lafont presents a notation for his Interaction Nets in which names can only be used linearly. It is unclear to what extent this notation can be thought of as a *programming language* rather than as a direct notation for reduction rules similar to the graph expressions I have used above. (I will have more to say about Interaction Nets in the section on other graph reduction systems that follows.)

In [Baw84], I proposed a programming language with only linear names. At the time I was still reluctant to introduce nondeterminism into the formalism, and so each vertex was restricted to have only one active terminal which trivially renders all grammars preclusive. This ruled out the kind of `Copy` vertices I have used here to support nonlinear names. Lacking any other way to support nonlinear names, I declared them illegal. Since I'm now reconciled with nondeterminism, I no longer advocate this approach to programming languages.

Backus, in his famous paper [Bac78], identifies names as one of the problematic features of current programming languages. Interestingly, he doesn't make any direct connection between names and the now-famous von Neumann bottleneck—he is interested in a different problem caused by the presence of names: the consequent need for an abstraction mechanism and the difficulties that abstraction would cause his proposed algebra of programs. The language he advocates, while perfectly free of names, still allows references to be duplicated through the use of data structure selectors. The resulting programs are still as full of nonlinearities as those written in any other language. I would argue that names are actually much more central to the problem he is trying to address, for it is through nonlinear naming that bottlenecks are formed.

Back in section 1.1.3 I pointed out that the standard techniques for stack allocating continuations work precisely because references to continuations are treated linearly in any programming language that does not allow continuations to be named. This may well be the oldest unconscious application of the notion of linearity in computer science. With the advent of compilers that translate programs into a continuation-passing style intermediate code [Ste78, KKR+86], compiler writers are forced to become more aware of the fact that there is something lost in that translation. Of course, they have not yet recognized that the missing information is in fact an instance of a much more general class of information, namely linearity.

Linearity pops up all over the place once you start looking for it. For example, in [HW91] the authors advocate a "swapping style" of programming that treats values in a linear fashion. They argue that the usual "copying style" causes software engineering problems. Even if you don't agree with their argument,[1] they have put their finger on nonlinearity as something worth thinking about.

Another example: In [Wat89] Waters develops a theory of "optimizable series expressions" that are written as if they manipulate arrays, but that can be compiled into efficient loops. Not surprisingly, it is the occurrences of nonlinear names that cause all the trouble. Expressions free of nonlinear names are trivially optimizable, while other expressions require more work to determine whether or not they are optimizable.

These last two examples are not unusual—they are representative of many other similar examples where nonlinearity has been making itself felt for years.

7.1.2 Graph reduction

There is not a strong connection between linear graph reduction and most other graph reduction systems, because the key notion of linearity is generally missing from more traditional systems [Tur79, Pey87]. In traditional graph reduction systems all vertices represent either values or expressions that will soon normalize into values. When linear graph reduction is used to model a real system, such as the Scheme system constructed above, vertices are used for a wide variety of purposes beyond just representing values. (consider `Copy` and `Return 1` vertices, or the vertices that represent continuations.) The only real commonality is that in all kinds of graph reduction there is an emphasis on having a *simple* execution model.

There are some other graph reduction systems that are linear. Both Janus and Interaction Nets are explicitly based on such systems. Lafont's illustrations even resemble the pictures I drew in chapter 2.[2] Both systems have an explicit directionality associated with the links in their graphs, and Interaction Nets imposes a type discipline on terminals. I have never felt a need for these additional constraints in my applications of linear graph reduction, although it is true that usually there is a directionality (and even a type discipline) implicit in the conventions adopted for building graph structure.

Interaction Nets arose out of attempts to bring functional and logic programming together with Linear Logic. (It is from this source that I originally borrowed the word "linear".) In Interaction Nets there is an interesting invariant (called **semi-simplicity**) on the form of the working graph, and all reduction rules must preserve it. This invariant serves to prevent deadlock in the unfolding computation. Semi-simplicity is a very interesting notion to explore because it succeeds in achieving a global goal (lack of deadlock) through purely local means (maintaining an invariant in each method). Given my characterization of state as a global property of the working graph, perhaps there is

[1] I don't entirely.

[2] The authors of Janus are particularly interested the graphical presentation of their graphs, and especially in animating them. They have also worked on animating both Lafont's Interaction Nets and my linear graph reduction system.

some variant of semi-simplicity that will prove relevant to the phenomenon of state.

Another graph reduction system with linear rules appears in Lamping's algorithm for optimal λ-calculus reduction [Lam90]. In this case a *specific* grammar is presented—that is, the set of methods is fixed and the program is encoded in the initial graph, just as it would be for a simple SK-combinator implementation. Optimal λ-calculus reduction is an interesting problem because nonlinear names cause expressions to be duplicated when β-reduction is applied. To avoid this duplication, and instead to *share* the graph structure representing the substituted expression, Lamping uses "fan nodes" in very much the same way as I have used `Copy` vertices. Lamping has resorted to linear graph reduction for precisely the reason I have advocated it here: in order to study a system with nonlinear naming (λ-calculus) he needs to expose the nonlinearities by using a linear model.

7.1.3 Distributed computing

In the area of distributed computing there is much work that bears some relation to mine. These systems can be roughly grouped according to how they name objects and locations in their distributed environments.

7.1.3.1 Explicit locations

One group of systems share with basic remote procedure call (RPC) [BN84] the fact that the programmer must explicitly name the network locations of remote objects and services. Some examples of such systems are NCL [FE85], REV [Sta86], NeFS [Sun90] and Late-Binding RPC [Par92]. These systems generalize RPC by allowing the programmer to specify an expression to be executed and the location to execute it. Typically a new keyword such as `AT` is added to the language and the programmer writes something like:

`(at` *location expression*`)`

The Mercury system [LBG+88, LS88] is not as general as these others, but it is targeted at the same performance problems with RPC. Mercury's call-streams support a form of remote execution where the control structure continues to execute locally, generating a stream of instructions, which are sent to the remote site, which executes them and returns a stream of results.

All these systems address the performance problems of pure RPC (including the "streaming problem" I mentioned in chapter 1) by allowing a process to move to a more appropriate location for the duration of some chore, potentially reducing multiple network round trip delays to a single round trip delay. I have addressed the same problems using demand migration, which avoids burdening the programmer with keeping track of explicit locations. (Of course, demand migration relies on heuristics, so it may sometimes perform worse than a well tuned system in which the programmer explicitly specified the optimal locations.)

In [Par92], Partridge presents a process migration scheme that he proves will always perform optimally in terms of network delay. Unfortunately, his proof only applies when there is a single task—his observation is of no help in deciding how to migrate data that may be shared by multiple tasks. My demand migration heuristics are designed to deal with precisely this problem.

None of these systems deals with tail-recursive calls properly (the "continuation problem" from chapter 1), although Partridge is aware of the deficiency in his system. My system has no problems with tail-recursive calls.

7.1.3.2 Location independent object names

Many distributed operating systems support a location independent naming mechanism for network objects. Examples are ports in Mach [YTR+87], UIDs in Chronus [STB86], and links in DEMOS/MP [PM83]. These systems all support an object oriented interface to the network objects referenced by such names, where the programmer manipulates objects by sending them messages. Messages are automatically routed to wherever the object is actually resident. Messages can include the names of other network objects, and such names will be usable by the recipient.

None of these systems explicitly address the performance problems of RPC, as the previous group did, but the use of location independent names does permit them to migrate tasks around to equalize processor load or to bring tasks closer to the objects they manipulate. In principle these systems could migrate a task whenever it was about to send a message to a remote network object—moving the task to the object so that the actual message send was always local. This would minimize the number of network trips. However, the tasks themselves are too unwieldy for implementors to take this option seriously. Instead, task migration is viewed as more of an occasional or periodic resource management problem.

The mechanisms necessary to make such location independent *nonlinear* naming work in these systems carry more overhead than light-weight links I used to support linear naming.

7.1.3.3 Unified naming

All the systems in the previous group use a different mechanism for naming remote objects than they do for local objects. A reference to a local object is typically a simple program variable, while a reference to a remote object is some distinctly different device that must be manipulated using some special facilities. In order to unify local and remote naming completely, remote naming must be accomplished from within the programming language.[3]

The Actor languages [Agh86], applied to distributed computing, are intended to be languages with such a unified naming system. A distributed Actor engine would look very much like my system. I could have chosen an Actor language instead of Scheme as my source language, but there would have been no real advantage in this, given that naming in an Actor language is no more or less linear than it is in Scheme.

Janus [SKL90] is also intended as a language for distributed computing with the requisite unified naming system. Furthermore its authors are aware of the benefits of linearity, and so Janus only supports linear naming. Janus is also a good candidate as a source language in place of Scheme—but I wanted to show what it was like to expose nonlinearities in a programming language by translating it into the linear graph reduction model, and that wouldn't have been possible using a language that had no nonlinearities!

I hope that the implementors of Janus or any of the Actor languages will derive some benefit from my experience applying linearity to the problem of building a similar system.

7.1.4 Programming language semantics

The programming language used in this dissertation is *almost* ordinary Scheme. The only exception was the addition of the FUTURE construct in order to introduce concurrency. The proper semantics for FUTURE is currently a subject of debate within the Scheme community. The issues revolve around futures and continuations created by expressions such as

```
(future
  (call-with-current-continuation
    (lambda (cont)
      ...)))
```

If such a continuation is invoked in an unexpected way—more than once, or from within some foreign process—what should happen to the corresponding future? I have adopted the semantics found in MultiLisp [Hal84], where the first invocation of the continuation always determines the value of the corresponding future and additional values are simply discarded.

Another possibility is described in [KW90], where additional values are returned from the original FUTURE form. This alternate behavior would be easy to accomplish within my existing system.

Not so easily accommodated is the behavior of futures in MultiScheme [Mil87]. MultiScheme supports explicit "tasks", which are used to control resource allocation. Tasks are intimately related to futures because the MultiScheme garbage collector uses tasks to determine what processor resources can

[3] It would also be possible to to accomplish the same goal at the low level by building hardware that treated local memory addresses and remote references on an equal footing.

be reclaimed in the event a future becomes garbage. When a continuation such as the one created by the expression above is invoked, the value is returned to the future associated with the current task, even if that future is not the future that was created to correspond to the continuation. This simplifies a resource management problem but in return it horribly complicates the semantics of futures. The linear graph reduction approach of garbage collecting disconnected components addresses the same problems without the troublesome explicit tasks.

7.1.5 The `target/tail` protocol

The following method originally appeared in section 2.4 to demonstrate how a tree-climbing `Set Car` message allows a `Cons` to be mutated:[4]

```
(graph (0 1 2 3 4)
  (<Set Car> target:5 tail:2 cont:3 new:4)
  (<Cons> 5 car:0 cdr:1))
(graph (0 1 2 0 4)
  (<Cons> 2 car:4 cdr:1))
```

In effect *two* continuations, carried by the `cont` and `tail` terminals, are delivered to the site of the computation. The `cont` continuation is to receive the result of the call, and the `tail` continuation is to receive the next version of the mutable object.

The same protocol for achieving mutability is employed in Actor systems [Agh86]. The language used to specify actor behaviors typically has two commands, `reply` and `become`, which correspond to the actions of returning values to `cont` and `tail` respectively. This protocol also appears in [Her90] as part of a larger program to construct concurrent data structures from sequential specifications. It is also essentially the same technique used in [Hen80] to construct state-like behavior from within a functional programming language. This protocol is the natural outcome of the most obvious approach to state in a functional framework: view each new configuration and output as a function of the previous configuration and inputs.

In the linear graph reduction Scheme system described in this dissertation, the fact that objects are implemented using this protocol is not in any way exposed to the user, as it is the the other systems just mentioned. However, this would not be a difficult feature to support. All that is needed is a new special form, similar to `LAMBDA`, that puts the the act of reconstructing the object after a call under the control of the user (instead of having the compiler supply the reconstruction automatically, as it does for the procedures created by `LAMBDA`).

[4] This method uses the alternate, simpler, continuation protocol where a value is returned to a continuation by direct attachment.

7.1.6 Thinking about state

Dixon [Dix91] approaches the phenomenon of state in a manner that is similar in spirit to mine. Dixon is interested in improved programming language support for "embedded" programs—programs that are part of, and must interact with, a larger computational environment. He develops an alternative approach to programming language semantics that focuses on the way the rest of the system is perceived by an embedded program, rather than attempting to describe the system as a whole. This is the essence of the approach to state I adopted in chapter 6.

It may be that his language Amala would be a good place to begin the search for better metaphors for state that I advocated at the end of chapter 6. Amala has a distinctly imperative flavor that I find hard to reconcile with the kind of fine control I imagine would be required of a language with a true understanding of state. Still, it might prove enlightening to work on translating Amala programs into linear graph grammars to see what might be revealed.

7.2 Future research

This section contains a sampling of research directions that build on the work described here.

7.2.1 Linearity

It is not hard to find new places where the notion I have called "linearity" appears. Once you get the knack of thinking about whether a name is being used linearly or not, it becomes the natural approach to any problem in which there is a bottleneck to some resource, a difficulty with state, a question of garbage collection, or any situation where multiple references to an entity may be involved. Since naming systems appear in almost every corner of computer science, and since such systems rarely restrict the names so as to keep them linear, there is a lot of territory that needs to be revisited with linearity in mind. I hope that after reading this dissertation the reader is inspired to watch out for this ubiquitous phenomenon in his own area of interest.

7.2.2 Linear graph reduction

In this dissertation we have seen linear graph reduction used as

- a compiler's intermediate representation for a program,
- a virtual machine (for distributed computing), and
- a mathematical tool for modeling real systems.

Each of these uses can be further explored on its own.

In the area of compilers, a lot of interesting work on static analysis and optimization of linear graph grammars is possible. The simulation technique I employed in chapter 3 is only the first step in this direction. An example of another technique is the formation of what might be called **macro-vertices**. A macro-vertex is a single vertex which can be used in place of a complex subgraph that the compiler guesses will occur frequently at run-time. This technique could be used, for example, to eliminate the cycles that are commonly introduced by the translation of Scheme's `LETREC` construct.

A related technique is to compute the set of pairs of terminal labels that can possibly be found on opposite ends of a connection at run-time. This can be done using a straightforward transitive closure algorithm over the set of known methods. This information could be used to select likely candidate subgraphs for macro-vertices, or to design special case run-time representations for certain connections.

The linear graph reduction virtual machine has the advantage of being extremely simple. This makes it easy to build reduction engines for it, but more work can be done to make such engines efficient. For example, the existing compiler compiles every method into code that operates at the level of rearranging vertices and connections, but it should be possible for the compiler to detect many situations in which the next reduction will invoke an arithmetic primitive, and so the compiler can instead directly output native code for performing arithmetic. Similarly, it should be possible to generate native control structures (dispatches and conditionals) instead of always relying on the general method lookup.

As a mathematical tool I'm sure linear graph reduction has more surprises still in store. The results presented in chapter 6 were discovered after only a fairly shallow examination of the category theory of linear graphs.[5] Since linear graph reduction is intended to fill the role of λ-calculus when linearity is important, any place that λ-calculus is used is a candidate for reinterpretation using linear graph reduction. For example, it would be interesting to construct the equivalent of denotational semantics in the linear graph reduction world. I don't know what this would look like, but perhaps it would reveal a deeper connection to the familiar algebraic notion of linearity.

7.2.3 Programming languages

Naming issues are central in programming language design, but existing languages have not been designed with linearity in mind. Programming languages constructs can be designed which promote linearity. For example, consider the Scheme procedure:

[5] You should be glad I stripped out the category theoretic overhead before presenting it!

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

The variable `X` is not linear since in the case where it's value is not the empty list it will be used three times. We can add a `DISPATCH` special form to Scheme that combines type dispatch with appropriate destructuring:

```
(define (append x y)
  (dispatch x
    ('() y)
    ((cons a d) (cons a (append d y)))))
```

(`DISPATCH` looks a lot like the kind of pattern directed invocation one finds in ML or Prolog.) Now there are four variables (`X`, `Y`, `A`, and `D`) which are all linear. This in not just hiding the nonlinearity inside the implementation of `DISPATCH`, for `DISPATCH` can be translated directly into the method dispatch inherent in linear graph reduction.

There are also alternate strategies to be explored for translating programming languages into linear graph reduction. The compiler described in chapter 3 was designed to be faithful to the well-known, essentially sequential, semantics of Scheme. Alternate languages are possible that have a more parallel semantics—languages in which all the expressions in a procedure application are evaluated in parallel. I described such a language in [Baw86]. The chief difference between the two languages turns out to be the protocol used for continuations: the language in [Baw86] lacks `Return 1` vertices, instead continuations are simply attached directly to the returned value.[6]

Finally, do not overlook the possibility of using linear graph reduction for pedagogical purposes. When explained in linear graph reduction terms the `FUTURE` construct is particularly clean, and the mechanics of a call to `CALL-WITH-CURRENT-CONTINUATION` are easy to see. Pictures of linear graph structure can capture more than the traditional "box-and-pointer" diagrams, because linear graph structure also represents the executing processes (e.g. using `Call` vertices and continuation vertices).

7.2.4 Garbage collection

Reclamation of the resources devoted to linear graph structure has many different aspects, only some of which have been explored in the system I built. The existing system uses `Drop` and `Copy` vertices to achieve the same effect as a reference count garbage collector. This works at both compile-time and run-time. The existing system also collects unobservable disconnected components at compile-time during simulation, but at run-time such structure is not

[6]Two methods that use this simple protocol were shown on page 30.

detected—storage and processor resources can be wasted. Detecting unobservable disconnected components in the distributed environment is an interesting variant on the distributed garbage collection problem.

A related resource issue is the detection of **stymied subgraphs**. A stymied subgraph is a subgraph bounded by inactive terminals that contains *no* redexes. The simplest example is a pair of monoactive vertices connected by their active terminals, to which no method applies. Such subgraphs are connected to the rest of the working graph, but are incapable of interacting with it. It is unclear exactly what to do about stymied subgraphs, but one possibility is to reclaim the storage devoted to them. (Another possibility is to treat the appearance of a stymied subgraph as an error.) Detecting stymied subgraphs, even in the non-distributed case, is an interesting algorithmic challenge.

There might be further generalizations of the notion of stymied subgraph that rely on an analysis of the universe of methods that is deeper than the simple classification of terminals into active and inactive.

7.2.5 Distributed computing

At the end of chapter 5 I listed a number of things that could be done to make the existing graph reduction engine more practical. Many of those tasks are suitable directions for future research.

The most important and difficult item on that list is probably fault tolerance. A brute force approach to fault tolerance is to equip every agent with its own stable storage to store a recent checkpoint of the local part of the working graph and a log of activity since then. Careful attention to the order of events such as stable storage writes, message transmissions, and message acknowledgments, can probably make such a scheme workable. However, it is certainly unnecessary for *every* agent to have stable storage—there is interesting ground to be explored here.

I imagine that there might be people who are somewhat skeptical that the local graph reduction part of the existing system can ever be made as efficient as the output of the best C compilers. Such people may be tempted to reject this entire exercise on those grounds. I invite those people to consider the possibility that the linear graph reduction view of the world could be used as *just* a network protocol. Inner loops could still be coded up in C or assembler, but when it came time to interact with the network, everything (pointers, stack frames, record structures, etc.) would be converted into linear graph structure. Linear graph structure would be the network lingua franca. Such a system would still benefit from linearity in the two ways that the current system does (i.e. cheap links and workable demand migration).

7.2.6 State

This is an area where I expect linear graph reduction to produce more surprises. The perspective on state presented in chapter 6 is radically different from the way state is normally viewed, and perhaps a little difficult to get used to, but its embedded observer approach to the phenomenon is really the only approach that can get a firm grip on what state really is. It is possible that the results of chapter 6 can be phrased in terms of some other nonlinear model of computation, but I suspect that the difficulties in dealing with nonlinear references would overwhelm the rest of the argument and obscure the fundamental insight.

In the area of state the research direction most likely to bear fruit is probably the search for better programming language metaphors and constructs discussed at the end of chapter 6. Possibly this can be approached by designing a programming language from the ground up based on linear graph reduction. If this approach proved successful, the next step would be to consider adding the new ideas to existing languages.

Perhaps related to such programming language research is the search for better ways of managing state in a distributed environment. (This area is one of the points of contact between the two applications of linear graph reduction described in this thesis.) Better programming language metaphors could also be useful metaphors for thinking about distributed state. Or perhaps the characterization of state will help in the development of a fault tolerant linear graph reduction engine.

Many compiler-oriented applications for the new characterization of state suggest themselves. For example, imagine a compiler that notices which methods have right hand sides that introduce cycles into the working graph, and which methods have left hand sides that make the grammar non-preclusive. Such a compiler may be able to use this knowledge to recognize when the apparent use of a side effect is not visible outside of some block, or is in some other way not an "essential" side effect. Perhaps such non-essential side effects can then be optimized out of the program. Such an analysis might be useful for even a FORTRAN compiler to perform, if it can learn something useful about the program.

Finally, the characterization of state needs to be pushed further. The insight that systems in which state occurs are systems which depend on (untestable) global properties of the system as a whole, doesn't yet feel complete to me. More investigation is needed into the nature of those global properties. One possible approach is to think about the Interaction Net notion of semi-simplicity, which is a different kind of global property. Perhaps it would be fruitful to investigate methods that fail to to maintain semi-simplicity as an invariant, or the interaction of semi-simplicity with linear graph homomorphisms.

7.3 Contributions

Finally, a review of the contributions made by this work:

Linearity. This notion was key to everything that followed. Linear names can only be used once, and thus cannot be used to create more than one outstanding reference to an entity. Thus, linear naming is cheaper to support than fully general naming, and it is also easier to reason about.

The linear graph reduction model. A simple computational model in which all references are linear. Translating a program onto a linear graph grammar can expose nonlinearities, just as translating it into continuation-passing style λ-calculus can expose unnamed quantities.

Translating Scheme into a linear graph grammar. Scheme is representative of a typical sequential programming language. I demonstrated how arbitrary Scheme programs, including those that use side effects, can be compiled into a linear graph grammar. I also demonstrated how to support the `FUTURE` construct.

To demonstrate the power of linear graph reduction I presented two applications that both build on this translation: a distributed programming environment, and a new theoretical characterization of state.

Linear graph structure as a universal distributed representation. For the distributed programming environment, all run-time structures (continuations, record structures, procedures, etc.) are represented explicitly using linear graph structure. This ensures the proper treatment of continuations in tail-recursive procedure calls.

Linear network references: links. By taking advantage of the properties of linear naming I was able to build a distributed programming environment in which cross-network references are cheap. Cheap cross-network references permit data structures to be highly portable.

Demand migration heuristics. Linear naming also facilitates the construction of heuristics that migrate tasks without requiring explicit guidance from the programmer. This works in a way that is analogous to demand paging.

An embedded approach to state. In order to properly approach the phenomenon of state it is important to think about how state is perceived by observers embedded in the system itself.

A characterization of state. State occurs when some locally indistinguishable configurations of a system may evolve in unintended ways. Furthermore, no test the system can perform internally can determine that it is properly configured. This explains why programming in the presence of state is difficult.

The flawed object metaphor. Since state is not a phenomenon that can necessarily be localized, the usual object metaphor for state may be fatally flawed.

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Bac78] 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.

[Baw84] Alan Bawden. A programming language for massively parallel computers. Master's thesis, MIT, September 1984. Dept. of Electrical Engineering and Computer Science.

[Baw86] Alan Bawden. Connection graphs. In *Proc. Symposium on Lisp and Functional Programming*, pages 258–265. ACM, August 1986.

[BN84] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[DG73] B. DeWitt and N. Graham, editors. *The Many-Worlds Interpretation of Quantum Mechanics.* Princeton University Press, 1973.

[Dix91] Michael Dixon. *Embedded Computation and the Semantics of Programs.* PhD thesis, Stanford University, September 1991.

[FE85] Joseph R. Falcone and Joel S. Emer. A programmable interface language for heterogeneous distributed systems. TR 371, Digital Equipment Corp. Eastern Research Lab, December 1985.

[GL86] 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.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hal84] 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.

[Hen80] 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.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. Symposium on Principles and Practice of Parallel Programming*, pages 197–206. ACM, July 1990.

[HW91] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proc. of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 105–112. ACM, August 1986.

[KW90] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *Proc. Symposium on Lisp and Functional Programming*, pages 176–184. ACM, July 1990.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[Laf90] Yves Lafont. Interaction nets. In *Proc. Symposium on Principles of Programming Languages*, pages 95–108. ACM, January 1990.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. Symposium on Principles of Programming Languages*, pages 16–30. ACM, January 1990.

[LBG+88] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler, and William Weihl. Communication in the Mercury system. In *Proc. Hawaii Conference on System Sciences*, pages 178–187. IEEE, January 1988.

[Lie81] Henry Lieberman. Thinking about lots of things at once without getting confused: Parallellism in Act 1. Memo 626, MIT AI Lab, May 1981.

[LS88] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. Conference on Programming Language Design and Implementation*, pages 260–267. ACM, July 1988.

[Mil87] James Miller. MultiScheme: A parallel processing system based on MIT Scheme. TR 402, MIT LCS, September 1987.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In *Proc. First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 1986.

[Par92] Craig Partridge. *Late Binding RPC*. PhD thesis, Harvard University, January 1992.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1987.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proc. Ninth Symposium on Operating System Principles*, pages 110–119. ACM, October 1983.

[Pos80] J. B. Postel. User datagram protocol. Request for Comments (RFC) 768, USC/Information Sciences Institute, August 1980. Available from the ARPANET Network Information Center.

[Pos81] J. B. Postel. Transmission control protocol. Request for Comments (RFC) 793, USC/Information Sciences Institute, September 1981. Available from the ARPANET Network Information Center.

[RC92] Jonathan Rees and William Clinger. Revised^4 report on the algorithmic language Scheme. Memo 848b, MIT AI Lab, March 1992.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proc. North American Logic Programming Conference*, pages 431–446. MIT Press, October 1990.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 192–214. ACM, August 1991.

[SS78] Gerald Jay Sussman and Guy L. Steele Jr. The art of the interpreter or, the modularity complex. Memo 453, MIT AI Lab, May 1978.

[Sta86] James W. Stamos. Remote evaluation. TR 354, MIT LCS, January 1986.

[STB86] Richard E. Schantz, Robert H. Thomas, and Girome Bono. The architecture of the Chronus distributed operating system. In *Sixth International Conference on Distributed Computing Systems*, pages 250–259. IEEE, May 1986.

[Ste76] Guy L. Steele Jr. LAMBDA: The ultimate declarative. Memo 379, MIT AI Lab, November 1976.

[Ste78] Guy L. Steele Jr. RABBIT: A compiler for SCHEME (a study in compiler optimization). TR 474, MIT AI Lab, May 1978.

[Ste90] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[Sun90] Sun Microsystems, Inc. Network extensible file system protocol specification, February 1990. Contact nfs3@sun.com.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.

[Wat89] Richard C. Waters. Optimization of series expressions: Part II: Overview of the theory and implementation. Memo 1083, MIT AI Lab, December 1989.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. Eleventh Symposium on Operating System Principles*, pages 63–76. ACM, November 1987.