Linear Naming and Computation

Working Paper No. 1 October 1998

Linear naming: experimental software for optimizing communication protocols

Alan Bawden Harry G. Mairson

Abstract

We propose to design and implement significant new forms of procedure calling protocols, together with relevant supporting formal tools and implementation technology, and experiments that evaluate their feasibility and effectiveness.

Contents

1 Introduction

We propose to design and implement significant new forms of procedure calling protocols, together with relevant supporting formal tools and implementation technology, and experiments that evaluate their feasibility and effectiveness. This research program combines development of applications technology with fundamental research on programming language design. Previous exploratory research into the formal theory of *sharing*—a careful realization of naming mechanisms sometimes called "optimal reduction"—has yielded techniques for the efficient sharing and evaluation of both values *and* computation threads. We are convinced that it is time to push this research further in the direction of pragmatics and implementation.

Procedure calling protocols and their interaction with naming form the glue that holds complex computations together. "Linear naming" and "linear graph reduction" are the tools used to implement sharing. Linear graph reduction keeps careful track of the complicated relationships between objects that can be caused by sharing. This in-depth account of sharing has yielded many interesting theoretical insights, and we believe that it is now ripe to yield pragmatic new implementation technology—especially for *distributed* computation, where the costs of sharing can be particularly pernicious.

To this end, we intend to design and implement a new procedure-calling protocol incorporating the desirable semantics of call-by-value as well as the efficiency of call-by-name. The fundamental technology behind this protocol, a radical new implementation of shared computation, potentially applies to a wide range of systems, including compilers, theorem provers,[1] programming languages and network communications.

The implementation of this technology has significant implications for both local and distributed computation. The payoff in terms of local computation is a safer semantics where programs are less likely to hang inadvertently, with formal properties that are much easier to reason about, as in languages with delayed evaluation. However, the technology promises a greater efficiency due to the smart, demand-driven, and incremental evaluation of delayed objects, sharing both values and computation threads in the process. The payoff for distributed computation is a suite of network protocols based on *linear naming*—where heuristics similar to those for LRU paging are used, except in the domain of *processes* rather than in the domain of *data*—which permit efficient migration

[1]In a theorem prover, a logical equivalent of "procedure call" is an essential feature in simplifying the internal representation of proofs.

of processes to minimize network traffic.

These software innovations depend on the technology of *sharing computation*—not merely the ubiquitous sharing of values, but also the sharing of computation threads. Thus an integral part of this research is the development of language primitives for handling values and continuations in an entirely similar manner.

We plan to implement formal tools for designing and evaluating graph reduction rules, including a virtual linear graph reduction machine. We will experiment with the technology to build and analyze applications in the real-world domain of network management. In addition, we envision language design for the symmetric handling of values and continuations, based on value and continuation types, ensuring full composability with the assurances provided by type safety. We intend to implement compiler extensions to support these facilities, and to make an analytic comparison with the "monadic" implementations of continuation, exception handling, I/O, and state. We also plan implementation improvements to the current technology for incremental evaluation of closures, and benchmarked experiments to test a standard suite of graph reduction primitives versus customized ones that can be generated from a compiler.

This experimental research program combines development of applications technology with foundational research on programming language design. As such, it is dedicated to making fundamental advances through innovations in software, where this progress requires the diverse, multiple talents of the research team members. The project builds on the maturation of earlier, successful research that was primarily conceptual in nature. We believe it is time to push such previous exploratory research further in the direction of experimental software development. The ideas are great ideas, and have a genuine conceptual beauty. We really like them, and believe they have something to contribute to computing practice. The best way to demonstrate that is to put them to work.

The impact of the work is a new paradigm for procedure calling in local and distributed computing, with the promise as well of providing efficient, interoperable representations in related software, such as compilers and theorem provers. The technology supporting this development promises an improved functionality and semantic clarity, while not sacrificing efficiency concerns.

1.1 Implementation and experimentation goals

We intend to develop a related group of software artifacts in this research program, together with a careful analysis of them. These include:

- Automated tools for designing and evaluating graph reduction rules, to aid the system designer in conceiving and analyzing reduction rules.

- A suite of network protocols supporting computing with linear references.
- A distributed implementation of a virtual linear graph reduction machine.
- New techniques for compiling linear graph reduction rules into efficient runtime code.
- New language constructs for handling control flow, with relevant compiler extensions.
- An implementation of "symmetric building blocks" for manipulating continuations, state, I/O, etc., including a comparative analysis with monads [Mog91, Wad92], a standard functional programming solution to these problems.
- Experiments evaluating "standard" versus "customized" graph reduction rules.
- New techniques for optimizing the demand-driven incremental copying of closures that lies at the heart of our technology.
- Sample applications that test our technology in the real-world domain of network management.
- While this is not software, we continue independent (and separately funded) supportive work on the theory of optimal reductions.

1.2 Technical summary

Our main contribution will be the further design and implementation of a new procedure-calling protocol that incorporates the desirable semantics of call-by-value with the efficiency of call-by-name.[2] Achieving this goal depends on a technology of *sharing computations*—not merely the ubiquitous sharing of values, but also the sharing of computation threads. The technical challenge that needs

[2]The protocol evaluates arguments at most once (as in call-by-value), and then only if they are needed (as in call-by-name). Procedure arguments should not be repeatedly evaluated if they are repeatedly used—a failing of call-by-name, and particularly so when these arguments are themselves procedures. In this latter case, a naive "lazy caching" of arguments does not work: to see why this is the case, consider `P(f)= pair(f M, f N)`. We can imagine code for the function `f` being shared—but how do we "unshare" its two *uses*, namely the respective applications to `M` and `N`? A more complex example where both sharing and unsharing is needed can be seen in the evaluation of the Scheme code `((lambda (x) ((x z)(x t))) (lambda (y) ((lambda (z) z) y)))` using the substitution model [Lév80, AS85].

to be surmounted to make this technology work is to get these two kinds of sharing to work in tandem. Think for a minute about how much effort has gone into static program analysis to get effects like the functionality of call-by-name, with the efficiency of call-by-value. For example, *strictness analysis* was invented to optimize "strict" programs in lazy languages that loop only if the argument loops, allowing the special-case use of call-by-value to gain runtime speed. In contrast, part of the idea here is to invent an efficient program evaluator that does the right thing *naturally,* without analysis.

Closely linked to this sharing technology is the development of language primitives for handling values and continuations in an entirely symmetrical manner, providing the essence of concurrent programming. Such asynchronous computation facilitates the idea of "network futures," and it promises a better semantics and pragmatics for distributed processes that need not synchronize with the arrival of their input data until that data is actually used. When the input data is itself a procedure, and only parts of the procedure has been communicated, such processes may proceed *incrementally.*[3] In addition, the improved semantics is a tool in any formal-methods approach to software quality, where improved reasoning about program behavior may facilitate development of safety-critical systems.

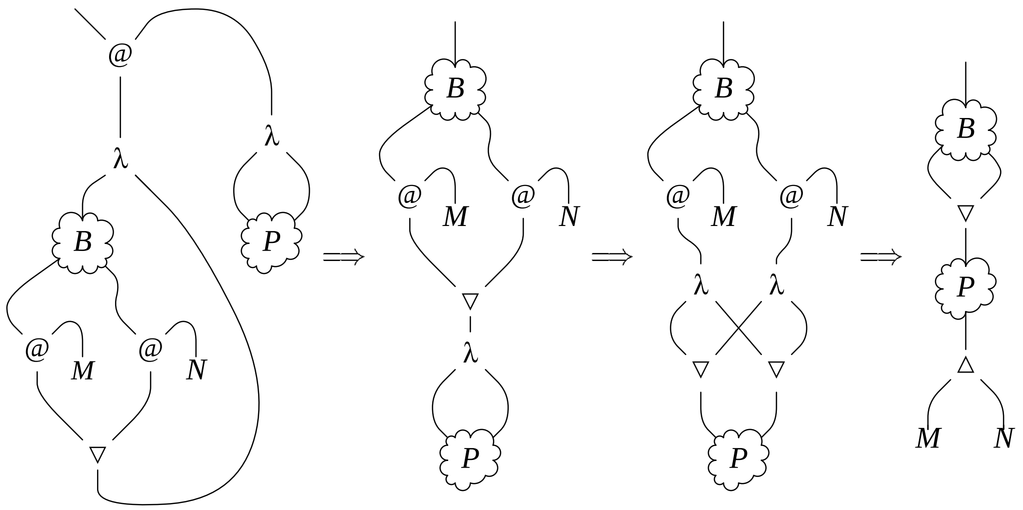
Our technology promises a viable alternative to RPC among other known remote procedure invocations, and solves standard problems with network communication that we describe below, including the so-called *continuation* and *streaming* problems. In a distributed environment, parts of an evolving computation 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. It is therefore natural that something which claims to be an improved way to think about naming should find application in distributed computing. Proper attention to linearity in naming 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.

What do linearity and optimal reduction do? Amidst the claims being made here, we need to say plainly what our technology actually does. In its simplest form, we conceive of computations in the form of graph reduction, where answers are produced by simplifying (graph) data structures. A standard kind

[3]We're as tired of waiting for Java applets to download as you are!

of reduction would involve ternary *application nodes*, which link a *procedure*, an *argument*, and a *calling context*, and ternary *procedure nodes*, which link a *calling context*, a *procedure body*, and a *parameter*. Procedure invocation is explained entirely as a *local* (and, as we conceive, distributed) graph simplification, where the parameter is connected to the argument, and the calling context of the application is connected to the procedure body. In implementation terms, *linearity* refers to the bounded locality and modularity of this procedure call.

Because procedures often have many references to the parameter, some *sharing* mechanism is needed also, so we include nodes that "fan out" a shared datum. What then is to be done when a procedure is being shared in two different calls? The usual solution, employed in usual complex implementations of such software systems with procedure invocations, is to make two *copies* of the procedure, one for each calling context. Instead, we implement the sharing in the two calls as the sharing of the procedure body, combined with the *unsharing* of the parameter. In other words, a program $\ldots p(M) \ldots p(N) \ldots$, with multiple, shared calls to procedure p, would result in the two calling contexts sharing the *code* for p, while an ensuing reference to the *parameter* would be (un)shared by the two arguments M and N. A picture of this situation might look like:



where B is the graph of the body of the program, and P is the graph of the code of the procedure p. In the last graph, P is shared from above by the two occurrences within B as well as being (un)shared from below by M and N.

It is possible to show that the careful implementation of this approach optimizes the number of procedure invocations. Making this sharing and unsharing technology work involves nontrivial algorithms and data structures. Furthermore, making it feasible requires more work on pragmatics.

1.3 Broad rationale: new directions in programming language design

There is no doubt that Java has breathed new life into the genuine relevance of language design to computing practice. For years, Lisp was criticized as the intelligent man's way of wasting computer cycles, but implementation technology for functional languages has improved tremendously, and now we see some of its characteristic features—such as implementation via virtual machines, and garbage collection of heap-allocated data—incorporated into the Java language that is currently the *lingua franca* of the World Wide Web. Interpreted languages are a natural solution to problems of platform independence, as is heap-allocated data for implementing high-level data structures. Many lessons are to be learned from this turn in language design, including:

- Formal methods are relevant to language design and practical applications: the functional metaphor, careful design of type systems, and object-oriented programming all grew out of logic and the logical design of programming systems. Such collaborative efforts show the real benefits of a genuine marriage of theory and practice.

- Leading candidates for formal methods that are interoperable across platforms include statically typed languages (programs mixed with proved assertions), and graph reduction (a high level implementation that can be interpreted on virtual machines).

- Language design need not, and probably should not, be of complete languages, but rather of innovative extensible language components that will be integrated in the next generation of languages. Major, brilliant language design efforts have not succeeded in attracting sizable user communities, while still contributing individual, essential language features and applications that have been incorporated elsewhere.

- Language features should be driven by applications and domain specificity: real-world demands can retroactively validate interest in what earlier might be viewed as a theoretically interesting, but practically infeasible functionality.

- Today's "practical" language features are yesterday's long-range research efforts, and today's language research needs to be justified and supported by its long-term relevance. Equally important is that the research world cannot compete with a monolithic software industry that can instantly commit many man-years to short-term projects. Therefore, experimental research devoted to "real applications" must choose its niche carefully, and in domains that industry is not ready to embrace.

John Reynolds wrote a quarter-century ago, "We are convinced that the major limitations of current languages are not due to careless choices among known design options which are possible. Long range progress will require new unifications, new generalizations, and above all, new ideas." Procedure-calling protocols provide a great case in point. We are all taught in school that procedure calls are by value, by name, and by reference. But this is not an immutable law of physics! Many other options exists, and we propose to provide and implement one of them: a *hybrid* strategy, called *optimal evaluation*, incorporating the semantics of call-by-name, and the efficiency of call-by-value.

2 Research plan

2.1 Why linear naming? What does linear graph reduction have that a programming language does not?

A programming language—and we typically think of the λ-calculus as the primordial such language—lets you write an identifier as many times as you like. A programming language makes no distinction between linear names and nonlinear names. For many purposes, this property 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 the language *directly* in terms of the low-level names from the memory system.

This property is what makes translating a program into "continuation-passing style" [Ste78, App92] 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 language into machine language is relatively simple.

As it happens, all the names introduced when a program is translated into the continuation-passing style language are linear. But nothing in the source programming language 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 style. It too exposes things that were previously implicit in the program. In addition to exposing continuations and intermediate values, it also exposes nonlinear naming (i.e., sharing). This exposure comes about for analogous reasons: just as continuation-passing style style does not directly support intermediate values and continuations, the linear graph reduction model does not directly support nonlinear naming (sharing).

2.2 Applications to distributed computing

2.2.1 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, by allowing data and acknowledgments to flow in both directions at the same time, but it is very difficult to duplicate the way a stream uses the network given only RPC. 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.[4] 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. 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 Java [GJS96] or a dialect of Lisp or PostScript [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

[4]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.

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. People working on distributed systems are starting to work with this idea now [HWW93, CJK95].

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 should move from place to place as needed, in order to collect together the information needed for the computation to progress.

2.2.2 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 (i.e., sharing) 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. In the rest of this section we 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. 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. The choice of which vertices to migrate is made using heuristics similar to those used for LRU paging.

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 in a graph). 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 continuation 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 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.[5]

Linearity facilitates these solutions by minimizing the cost of adopting a uniform representation for references. We *could* achieve 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 could solve the continuation problem and achieve location transparency. These nonlinear references would be difficult, but not impossible, to maintain across the network.

However, 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.

2.3 Applications to programming language design and implementation

2.3.1 Graph reduction and implementation independence

Large system design depends in part on construction of disparate, interoperable components that can talk to each other. As these components reflect explicit, implementation-dependent characteristics, it gets harder to link them together. This of course is what has rehabilitated the idea of virtual machines and of language interpretation in the age of the ubiquitous Internet. Graph reduction is a technology that sits "lower" than a high-level language, but "higher" than the level of machine implementation. As such, we think it is a good level to base interoperable, implementation-independent system design. But in addition, we have definite ideas how graph reduction needs to be and can be done efficiently.

2.3.2 Delayed, demand-driven copying of closures

One of the fundamental problems of graph reduction is how it has required the copying of closures. This is seen in the sharing of procedures called in different contexts, but also in the implementation of call-by-value procedure calling,

[5] On those occasions when the programmer needs to *expose* the location of an entity, a suitable linguistic mechanism can make it available. Such occasions do arise when writing programs that interact with the real world.

where arguments ("thunks") get copied—all with an implicit slowdown as copied procedure calls are repeatedly invoked. The technology of optimal reduction is intended to replace this thrashing with a delayed, demand-driven copying of closures, where only the minimum necessary structures is copied as the computation evolves. The methods to do this demand-driven copying are already known. But more pragmatic work needs to be done to put them into practice.

2.3.3 Formal methods: linear logic and implementing continuations

Types are a canonical lightweight logic for reasoning about programs via formal methods: a type system is a restricted proof system with a tractable decision procedure. For this reason, one sees much research in "Types for X-inference" where X is some operational characteristic. But the usual version of types is only telling half the story, because types usually talk about values, and not continuations (control threads). Especially as control gets more complicated and is more relevant to complex computations, we want type systems that talk about both.

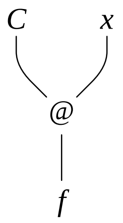
We always think of a function $f : A \to B$ as a transformer from a value of type A to a value of type B, but f is equally a transformer from a *continuation* (demand) of type B to a continuation of type A. This alternative view of computation is demand-driven, and is a lot more important when processes are arguments to procedures. In truth, procedures are called for *two* reasons: because an input value has been supplied, *and* because an answer is needed. Graph reduction captures this exactly, and does so in a typed framework.

Linear logic provides type information for this new perspective into what a procedure does, with new insights as well into how to write programs, and does so in a way that is very different from continuation-passing style [Gir87, Gir95]. This insight gives us a different view of *control threads* and into *naming*, where information about functions is considerably refined. In linear logic, a procedure of type $A \multimap B$ is one that accesses a datum of type A exactly once, and returns a datum of type B. This is alternatively written in the input/output-neutral form $A^\perp \wp B$, where $\wp$ is a special kind of "or"—the procedure can either return a *request* for an input of type A (denoted $A^\perp$), or output a *value* of type B. This neutrality reflects the fact that procedures transform control threads synonymously with their transforming values. A *call* to such a procedure has type $A \otimes B^\perp$—the $\otimes$ specifies a *pair* of data, where A is the type of an input value, and $B^\perp$ is a request for a datum of type B.

Many procedures use their inputs multiple times or not at all. We write $!A$ for the type of such input data, where the procedure now has type $!A \multimap B$ to refine the usual function type $A \rightarrow B$. Graph reduction is an operational stratum that captures these ideas: its notion of demand-driven copying fits hand-in-glove with the logical ideas inherent in linear types.

2.3.4 Symmetric language design

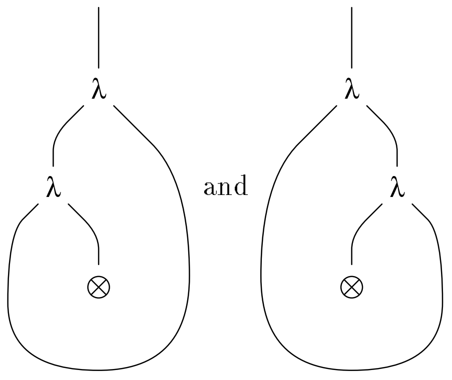
Following the above straightforward intuition of the "symmetry" of procedures, we recall research by Filinski, where he tried to construct a *symmetric language* where functions can abstract over either [Fil89]. He failed to get a true symmetry, and introduced an overly complicated language design to get his system to work, but the technology of graph reduction provides an avenue to a proper solution of his problem. For example, we can write $f \uparrow x$ to mean "apply function f to value x," and $C \downarrow f$ to mean "transform continuation C with function f." In the notation of graph reduction, we have:



Dually, we have abstraction over values and over continuations: $x \Rightarrow E$ is a function that inputs value x and returns value E, where E may have references to x; similarly, $y \Leftarrow C$ is a function that inputs continuation y and returns continuation C, where C may have references to y. Consequently, functions are just transformers of values (at the "domain" end) and continuations (at the "range" end). We now have two kinds of function calls, for value and continuation transformers respectively: $(x \Rightarrow E) \uparrow F$ reduces to $E[F/x]$, and $D \downarrow (y \Leftarrow C)$ reduces to $C[D/y]$.

On the other hand, graph reduction is *completely neutral* in its transformation of values and continuations: in graph reduction, there is only one kind of abstraction, and only one kind of application, and only one kind of function call.

For example, $x \Rightarrow (y \Rightarrow x)$ and $u \Leftarrow (w \Leftarrow u)$ are coded by the following graphs:



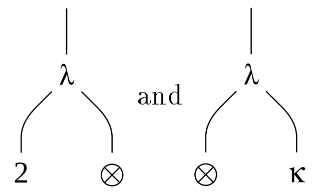
Notice that the plug $\otimes$ in the left graph coding $x \Rightarrow (y \Rightarrow x)$ is a "garbage continuation" that eats values in the course of a procedure call, while the plug in the right graph coding $u \Leftarrow (w \Leftarrow u)$ is a "garbage value" that gets eaten by continuations/contexts in a procedure call.

By drawing graphs, one can easily verify that $(x \Rightarrow x) \equiv (x \Leftarrow x)$ since the graphs are identical. More interestingly, we discover that $(x \Rightarrow (F \uparrow (G \uparrow x))) \equiv y \Leftarrow ((y \downarrow F) \downarrow G)$—they too have identical graphs. This last example shows that function composition $F \circ G$ can be described two ways.

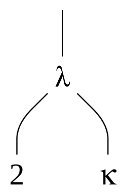
Now we can describe an essential problem: can these pieces be put together simply in a language? For example, how can one compose $x \Rightarrow 2$ and $y \Leftarrow \kappa$, where 2 is a constant value, and κ is a constant continuation? It can be written two ways: either $w \Rightarrow ((x \Rightarrow 2) \uparrow ((y \Leftarrow \kappa) \uparrow w))$ or $w \Leftarrow ((w \downarrow (x \Rightarrow 2)) \downarrow (y \Leftarrow \kappa))$. However, when we simplify the first expression using the "value" procedure call, we get $w \Rightarrow 2$, while when we simplify the second expression using the "continuation" procedure call, we get $w \Leftarrow \kappa$. Surely these two answers do not mean the same thing. The problem can be solved in an unsatisfying way by making the "value" world call-by-value, and the "continuation" world call-by-name. By this breaking of symmetry, evaluation results in one answer rather than in two.

But there is nothing wrong with two answers in a world with continuations, because there are multiple threads in the computation. As long as the threads do not mix in a contradictory way, there is no problem. (In contrast, we would not want $7 + 5$ to produce two different answers.) Let's look at the solution in optimal graph reduction, where call-by-value and call-by-name are synthesized into a hybrid protocol.

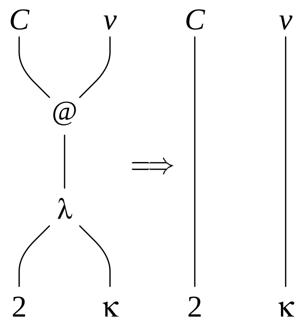
Since optimal evaluation graphs are a kind of hybrid of call-by-value and call-by-name, what does this example look like? $x \Rightarrow 2$ and $y \Leftarrow \kappa$ are coded as



When the paradox is repeated—and resolved— using graphs, the derived answer is



This is not a function in the ordinary programming-language sense! It is a "constant" function, but one that encodes *two* complementary constants: a constant value to be supplied to the calling continuation, and a constant continuation to be supplied to the argument of the procedure call! How can one decide which of the two constants was the truly "crucial" one? Why not instead keep both sides? Consider a procedure call using this function with continuation C and value v, where the procedure call gives you two disjoint computations:



In one computation, continuation C is given the value 2; in the other concurrently-running computation, continuation κ is given value v. Both threads of the computation are equally valid; rather than discard one of them arbitrarily, it makes more sense to keep them both, and there is no implied contradiction. But why is this so? A computation that has only one continuation/context (the "root") is *inherently functional*—only one answer has been requested. But multiple continuations correspond to multiple requests—there isn't anything "bad" about this situation.

In fact, this example motivates a perspicuous approach to *concurrent* computation that exists comfortably in the framework of functional programming, only one where *multiple answers* are allowed. The concurrency arises naturally and intuitively—it isn't a "wedding cake" feature among others that have been added, layer by layer, onto functional programming. It is, in contrast, an essential, intrinsic feature of the evaluation technology.

To make this technology work, quite a few real technical and implementation problems have to be solved. We need a clean language to describe the graph codings. We need underlying implementation technology to implement *sharing*, and the current state of the art is not good enough. And we need some clear benchmarks that clarify the functionality and expressiveness of this novel language. As an initial guide to providing suitable benchmarks, we note that the categorical idea of *monads* has had considerable success as providing the insight for clean software development of standard applications such as state, I/O, exception handling, and working with continuations. We believe that a symmetric language may well be an effective technology that provides an alternative to monads, and we would like to see how far we can push this novel technology.

3 Previous work

3.1 Some brief history

A brief history of this research area tells a lot about how theory and practice can meet to build good software artifacts. In the late 1970s, Jean-Jacques Lévy was working on the idea of shared computation in the λ-calculus [Lév80]: he knew what a solution would have to do, but lacked all the algorithmic pieces to put together a solution. The problem was really solved by John Lamping about 1990 [Lam90], using a graph reduction technology that was very similar to one invented by Yves Lafont, who was using it for other purposes [Laf90]. Simultaneously, Alan Bawden (one of the co-PIs) was working on linear naming, and built a real distributed Scheme system using the same kind of technology [Baw92]. The French scientists, motivated by theory, called Lamping an "autodidactic engineer," but the first such real engineer was Bawden.

3.2 Implementation experience

In the mid 1980s, Bawden was working on the seemingly unrelated problem of programming massively parallel SIMD computers, such as the Connection Machine [Hil85]. In order to control communications network contention problems in such machines, he was working with programming languages in which communications bottlenecks are prevented by the simple expedient of outlawing the ability to make copies of references. Exploring this space of computational mechanisms, Bawden developed the linear graph reduction model as the most elegant abstraction of the kind of systems he was trying to build. As it happened, the "Connection Graphs" he described in [Baw86] are almost indistinguishable from the "Interaction Nets" later described by Lafont in [Laf90]. We think there must be something very deep about linear graph reduction given that it can be discovered from such different directions.

During the late 1980s, Bawden got interested in the problems of computer networking. Looking for an application that would demonstrate the utility of linear graph reduction, he realized that it could be used as a basis for a distributed programming language implementation. By distributed the vertices among the processors on a network, and migrating subgraphs from processor to processor as needed in order to apply reduction rules, linear graph reduction could be used to improve on existing network technology such as remote procedure calls (RPC) [BN84] and network futures [LS88]. Bawden built the NLGR (Network Linear Graph Reduction) system to demonstrate these capabilities [Baw92]. Linear references made it easy to support cheap cross-network references and highly portable data structures, and linear references also facilitated the demand-driven migration of tasks and data around the network without requiring explicit guidance from the programmer.

For the last few years, Bawden has been working in industry in the areas of network management and the World Wide Web. In 1997 he came to Brandeis where he hopes to draw on his experience in the domain of network management to construct sample applications to further test the practicality of using linear graph reduction for distributed computing.

3.3 Foundational research

Mairson has committed substantial time and effort during the last few years working on the analysis of optimal reduction using graph reduction. What qualifies the graph reduction technology to have that name—what is so optimal about optimal reduction? In collaboration with postdoctoral researchers Lawall and Kucan, and Asperti at U. Bologna, he has investigated the algorithmic problems of graph reduction, shown why the technique can be considered efficient, and proved that the algorithm is correct [LM96, LM97, AM98]. Correctness is nontrivial, and a first-principles explanation has been lacking. Even the idea of implementation-independent cost for languages is not a worked-out area: defining the cost of a language via its compiler makes as much sense as defining the semantics of a language via its compiler—the latter was the point of denotational semantics. Meaning and cost should be implementation-independent and stand on their own. It was the growing insight of this foundational work, and the congruence of Bawden's closely-related implementation efforts, that have motivated the current proposal. Mairson attracted Bawden to Brandeis precisely to exploit the synergy between these related projects.

4 Research goals

We divide the work envisioned into development of technology, complemented by related science and experiment. The technology behind the software development is territory we are sure of, while the science and experiment component contains research and implementation problems we will have to work hard to solve.

4.1 Technology

4.1.1 Tools for designing and evaluating reduction rules

We want to have better computer-aided design when developing sets of linear graph reduction rules. Currently, paper-and-pencil (or whiteboard-and-marker) is the easiest way to think about these things, which limits the size of the examples you can work with. In contrast, we want to be able able to work with large sets of rules and large graphs, and to write succinct programs that manipulate and analyze such rule sets and graphs. Note that these tools do not have to be particularly efficient—it is a toolkit to be used for exploration, not a practical programming language implementation.

Working with paper-and-pencil does have the advantage that you can *see* the graph structure you are working with. It would be nice to have something that renders graph structure presentably on a computer monitor, perhaps animating the execution of rewrite rules, or directing execution using a mouse. While layout is a complicated problem for VLSI among other systems, we would like to have a try at such visualization, and there are geometric tools (e.g., the Delaunay system) that may be helpful here [CAL$^+$97].

4.1.2 A virtual linear graph reduction machine

The toolkit described above will be useful, but we want to use linear graph reduction as the basis for doing real computation. We want to build an efficient distributed runtime for linear graph reduction that can actually be used for practical programming.

Because we wish to operate in a distributed environment containing hardware platforms of a variety of different kinds, and because we wish to be able to introduce new reduction rules into the system and have them propagate over the network from processor to processor, we will develop a virtual machine for linear graph reduction. Programs written for our virtual linear graph reduction machine will describe linear reduction rules and graph structure in a concise byte-code language.

Since it is inefficient to actually do a sequence of graph reduction steps to evaluate a numeric expression, our virtual machine will be capable of describing more than just surgery on graph structure—we will need to support more mundane operations such as arithmetic. A set of linear graph reduction rules will be compiled into larger-sized chunks, so that operations like arithmetic can be performed directly, with no overhead from the graph reduction model at all. It may even be worth abandoning strict linearity at the local level, while keeping it at the network level as a way for the different processors to coordinate their actions.

As long as the processors behave *as if* they are performing fine-grained linear graph reduction, they are free to actually do something more efficient. This leaves us a lot of room for optimizations in the design of the virtual machine and in the compiler for that virtual machine.

Bawden's NLGR system did not have such a virtual machine. NLGR processors only communicated over the network about graph structure, never reduction rules. All network elements were preloaded with natively compiled versions of all the relevant rules. This required re-loading all processors for every program—not very practical. We intend to correct this deficiency.

4.1.3 A compiler

As hinted in the previous section, we will also be developing a compiler along with out virtual machine. The source language for this compiler will be some conventional programming language, augmented with new linguistic constructs. In particular, we expect to be able to define procedures whose parameters are the calling context as well as the argument, and the two should be treated in a virtually symmetric way.

The compiler's intermediate representation for programs will be in the form of linear graph reduction rules. The compiler will thus be a heavy user of some of the tools described in section 4.1.1. Much of the analysis and optimization performed on this intermediate representation will have as its aim the "chunking" described in the last section that will make for efficient local execution.

Bawden's compiler for NLGR performed only the simplest of possible optimizations: applying one rule to the right hand side of a different rule. This technique, which roughly corresponds to performing a β-reduction step, made a good start at the kind of "chunking" we will need to do, but there is room for improvement.

4.1.4 Network protocols supporting linear references

In order to support distributed graph reduction, we will need to migrate graph structure from processor to processor. Part of the job of the locally executing virtual machine will be to notice when a potential reduction is blocked because one of the two input vertices resides on a remote processor. When that happens, the virtual machine selects a subgraph of the local graph to migrate to the remote location. Selecting that subgraph is the job of the demand migration heuristics. Actually accomplishing the transplant efficiently will require support from a network protocol layer that can update all the effected edges with low overhead.

A network protocol for maintaining linear references was developed for NLGR. This protocol demonstrated that in terms of network traffic, it was quite cheap to maintain and update linear references. We will improve on that initial implementation in the following ways:

- We will integrate it into the adjacent network layers. Most importantly, the reliable messages that are required to transport migrating graph structure and reduction rules and also occasionally required by the linear reference mechanism itself, will be combined with the linear reference maintenance layer. This is because in both cases packet sequence numbers and retransmissions are used, and a great simplification should result from combining the two.

- We will address the issue of fault tolerance that was ignored in NLGR. When one processor fails, or the network becomes partitioned, the system must recover or fail gracefully.

4.2 Science and experiment

4.2.1 Test application: network management

To make sure this work stays grounded in reality, we propose to test our distributed programming language implementation by using it to construct some sample applications in the domain of network management.

In the Internet, network management is typically performed by a human operator using a program called a "manager." The manager runs on a single workstation and uses SNMP (the Simple Network Management Protocol) to probe and modify the state of "agents" that run in various devices deployed around the network (hubs, routers, gateways, modems, and other parts of the network infrastructure).

There are currently two widely recognized needs in network management: (1) a need to move more of the expertise of the human operator into the manager program, and (2) a need to move more of the manager's computation out into the agents. Network management is thus a domain where there is a growing pressure to move towards highly intelligent, truly distributed applications that operate over a very diverse collection of data.

Here's why we're interested in working in this domain:

- We believe our ideas are applicable to *any* distributed programming problem, so in particular, we believe they should work to do network management.

- There is vast amount of diverse information available via SNMP—information that is already deployed in the real world waiting to be exploited. It is also *realistically* deployed. There are good reasons for why that information is sitting out there in agents: it describes hardware and processes that are local to those agents. Programs that make use of that information are distributed for realistic reasons. It is a good test-bed for a programming language technology such as ours, which is designed to support heterogeneous distributed computing.

- The technique of "demand migration," which Bawden developed for NLGR, aims to minimize the number of network transits required to perform any particular computation. In a situation where the network is misbehaving, and thus some network management activity is needed, it is particularly important to avoid unnecessary network traffic.

  Since demand migration works entirely as a dynamic run-time optimization, it is a very flexible technique for optimizing usage of the network. It can take advantage of properties of the computation that are difficult or impossible to predict in advance. We can imagine a version of the demand migration heuristics that takes into account the current observed network behavior when deciding how much of a particular distributed management application to send where.

- The linear graph reduction virtual machine is a very *simple* model of computation. So it should be easy to support it in network agents that don't always have a large amount of memory or other computational resources.
- Bawden already knows a lot about this domain.

4.2.2 Symmetric language design: new methods for handling control flow

As described in Section 2.3.4, we want to design and implement language primitives for a "symmetric" language that allows computing with control and with values. As an application of this language, we want to construct symmetric "building blocks" for continuation calling, exception handling, and other standard facilities offered by monads in functional programming. We want to see how efficiently, in comparison, we can get our implementation to be.

4.2.3 Experiment: evaluating "standard" versus "customized" graph reduction rules

There are two different ways of implementing our basic idea: one involves a fixed set of graph nodes an operations which form the "target language" of compilation; another, forming the core of Bawden's NLGR system, involves a compiler that generates new kinds of nodes and reductions for every compiled program. We still want to know more about the relative efficiencies and pragmatics of these two approaches.

4.2.4 Optimizing demand-driven copying

The technology exists to do demand-driven copying, and we want to implement it. However, the underlying data structures that implement this copying need more work at the level of pragmatics. In particular, graph structures that can be copied are always enclosed in *boxes*—the essence of these is also known as a *closure* or a *thunk*. The optimal reduction works by incrementally copying these boxes, rather than copying an entire box. We want to be able to implement the proposed functionality of this incremental copying with data structures that ensure a genuine efficiency.

4.2.5 Independently supportive work: theory of optimal reduction

We wish to note that we carry on independent supportive work on the foundations of optimal reduction. One of the reasons we like this subject so much is its promise in the very related domains of both practice and theory. We have done considerable work in the past in algorithmic analysis of graph reduction and in the development of relevant cost models—in fact, it is this work that has encouraged us to look at practical application of the technology. More recently, we have worked on simple proofs of correctness of the relevant algorithms. We are currently interested in its relation to *full abstraction*—how one reconciles operational and denotational semantics for a programming language—and how it can be used to greatly simplify ideas from so-called "game semantics" that have been used recently with great success to attack the problem of full abstraction.

References

[AM98] Andrea Asperti and Harry G. Mairson. Parallel beta is not elementary recursive. In *Proc. 25-th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico.* ACM Press, New York, NY, January 1998.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Baw86] Alan Bawden. Connection graphs. In *Proc. Symposium on Lisp and Functional Programming*, pages 258–265. ACM, August 1986.

[Baw92] Alan Bawden. *Linear Graph Reduction: Confronting the Cost of Naming.* PhD thesis, MIT, May 1992. Dept. of Electrical Engineering and Computer Science.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CAL+97] Isabel Cruz, Michael Averbuch, Wendy T. Lucas, Melissa Radzyminski, and Kirby Zhang. Delaunay: a database visualization system. In *Proc. ACM SIGMOD*, pages 510–513. ACM Press, New York, NY, 1997.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[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.

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, Computer Science Department, 1989.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

[Gir95] Jean-Yves Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[Hil85] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[HWW93] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 239–248, July 1993.

[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.

[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.

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[LM96] Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 92–101, Philadelphia, Pennsylvania, 24–26 May 1996.

[LM97] Julia L. Lawall and Harry G. Mairson. On global dynamics of optimal graph reduction. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 188–195, Amsterdam, The Netherlands, 9–11 June 1997.

[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.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:1:55–92, 1991.

[Par92] Craig Partridge. *Late Binding RPC*. PhD thesis, Harvard University, January 1992.

[Pos81] J. B. Postel. Transmission control protocol. Request for Comments (RFC) 793, USC/Information Sciences Institute, September 1981. Available from `ftp://ftp.isi.edu/in-notes/`.

[Ste78] Guy L. Steele Jr. RABBIT: A compiler for SCHEME (a study in compiler optimization). TR 474, MIT AI Lab, May 1978.

[Sun90] Sun Microsystems, Inc. Network extensible file system protocol specification, February 1990. Contact `nfs3@sun.com`.

[Wad92] Philip Wadler. The essence of functional programming. In *Proc. ACM International Conference on Principles of Programming Languages*, pages 1–14, 1992.