A Graph Reduction Technique and an Extension to the G-Machine for Pure, Lazy Functional-Logic Languages (Preliminary Report)*

John Maraist
Universität Karlsruhe†

Frank S. K. Silbermann
Tulane University‡

Presented at the Third Workshop on Functional Logic Programming, January 13, 1994.

Abstract

We present a graph reduction technique for the implementation of functional-logic languages. Our technique, called *quasisharing*, attempts to provide a maximum degree of sharing between alternative narrowings while still admitting a reasonable compilation. Quasisharing resembles aspects of an algorithm of Lamping for optimal reduction in the lambda calculus. We also describe an extension to the G-machine which implements quasisharing.

1 Introduction

This report describes a graph reduction technique and implementation for lazy functional-logic languages. Rather than relying on backtracking for rebinding of logical quantifiers, we note alternative choices explicitly in the graph, and optimize space complexity by adapting a technique for optimal reduction of λ-terms due to Lamping [Lam90].

Source Language. Our source language is derived from Silbermann and Jayaraman's syntax [SJ92]. Source programs are a list of closed, supercombinator definitions matching *defn* in the grammar of Figure 1. We take K ranging over constructor names, and τ over the first order types[1] over which logical abstraction shall be made. Namely, abstractable types are neither function nor set types, but rather are sum and product types containing only abstractable subtype components. Aside from the usual reduction notions of β and δ in functional languages, we identify three notions of reduction peculiar to set manipulation.

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \$\mathit{name}\ x_1 \ldots x_n\ =\ \mathit{body} \\
\mathit{body} & ::= & \bigcup_{x\in\tau}\ \mathit{expr} \\
& | & \mathit{expr} \\
\mathit{expr} & ::= & \lambda x.\mathit{expr} \mid \mathit{expr}\ \mathit{expr} \mid x \mid \$\mathit{name} \\
& | & K\ \mathit{expr} \ldots \mathit{expr} \\
& | & \textbf{\textit{case}}\ \mathit{expr}\ \textbf{\textit{in}}\ (K^n\ x_1 \ldots x_n\ \text{->}\ \mathit{expr})^* \\
& | & \{\mathit{expr}\} \mid \mathit{expr} \cup \mathit{expr} \mid \phi
\end{array}
$$

Figure 1: Grammar for the source language.

First, σ reductions manipulate graphs of sets to reflect how set constructs interact with each other: that the empty set is an identity for the union node; that set abstraction should be distributed to

*This research was supported by contract LEQSF RD-A-19 of the Louisiana Board of Regents and by the Tulane Committee on Research.

†Most of this work was completed at Tulane University. Postal address: Institut für Programmstrukturen, Universität Karlsruhe, 76128 Karlsruhe, Germany. Email: maraist@ira.uka.de.

‡Postal address: Department of Computer Science, Tulane University, New Orleans, LA 70118, USA. Email: fs@cs.tulane.edu

[1] We do not address the details of a type system here, but rather assume some inference system, perhaps a standard Hindley-Milner inference system extended with rules for judgements regarding a new type constructor $\{\,.\,\}$:

$$
\frac{}{\Gamma \vdash \phi{:}\{\tau\}} \qquad \frac{\Gamma, x{:}\tau \vdash T{:}\{\tau\}}{\Gamma \vdash \bigcup_{x\in\tau} T{:}\{\tau\}} \qquad \frac{\Gamma \vdash T{:}\tau}{\Gamma \vdash \{T\}{:}\{\tau\}}
$$
$$
\frac{\Gamma \vdash T_1{:}\{\tau\} \qquad \Gamma \vdash T_2{:}\{\tau\}}{\Gamma \vdash T_1 \cup T_2{:}\{\tau\}}
$$

The restriction of abstraction to certain types will likely require some restrictions on the use polymorphism in these expressions. We leave a proper investigation of the properties of a suitable type system as future work.

beneath binary union, which will require explicit substitutions; and that we may have built-in operations over sets, in particular `$dist`[2],

$$
\begin{aligned}
&\texttt{\$dist}\ \phi\ f &&= \phi\\
&\texttt{\$dist}\ \{x\}\ f &&= f\ x\\
&\texttt{\$dist}\ (x \cup y)\ f &&= (\texttt{\$dist}\ x\ f) \cup (\texttt{\$dist}\ y\ f)\\
&\texttt{\$dist}\ (\textstyle\bigcup_S x)\ f &&= \textstyle\bigcup_S (\texttt{\$dist}\ x\ f) \quad .
\end{aligned}
$$

N-reductions manipulate relative set abstractions. The default evaluation mechanism for relative set abstraction resembles the reduction of list comprehensions used in many current functional languages [Pey87, e.g]. But unlike the treatment of list comprehensions, we distinguish abstraction over certain sets of first-order terms by taking the enumeration parameter as a logical variable quantified over the generating set, and reducing the body of such an abstraction as if that parameter had been instantiated by one of the ground terms in the generating set. This technique provides a logic programming capability comparable to first-order set abstraction [DFP86] without imposing any inelegant first-order restrictions[3]. N-rules are derived from equations partitioning sets (domains) in meaningful ways — when the application of some primitive function to a quantifier is not reducible to the same result for every element of the domain over which the quantifier is abstracted, the abstraction is divided into abstractions over smaller domains such that further reduction are possible. For types composed of a small number of tagged alternatives, say a type τ,

$$
\begin{aligned}
\tau = {} & K_1\ \tau_1^1\ \cdots\ \tau_1^{i_1}\\
& + K_2\ \tau_2^1\ \cdots\ \tau_2^{i_2}\\
& \vdots\\
& + K_n\ \tau_n^1\ \cdots\ \tau_n^{i_n}\ ,
\end{aligned}
$$

we rewrite expressions to consider each tagged alternative separately,

$$
\begin{aligned}
\textstyle\bigcup_{x\in\tau} M \ \rightarrow\ & \textstyle\bigcup_{x_1\in\tau_1^1} \cdots \bigcup_{x_i\in\tau_1^{i_1}} \left[\frac{K_1\ x_1\ \ldots\ x_{i_1}}{x}\right] M\\
& \cup\ \textstyle\bigcup_{x_1\in\tau_2^1} \cdots \bigcup_{x_i\in\tau_2^{i_2}} \left[\frac{K_2\ x_1\ \ldots\ x_{i_2}}{x}\right] M\\
& \vdots\\
& \cup\ \textstyle\bigcup_{x_1\in\tau_2^1} \cdots \bigcup_{x_i\in\tau_2^{i_2}} \left[\frac{K_2\ x_1\ \ldots\ x_{i_2}}{x}\right] M\ .
\end{aligned}
$$

For types composed of a large number of elements (such as, say, the integers), or for asserting a disequality between two quantifiers, we reduce one abstraction to a binary union of abstractions,

$$\bigcup_{x\in\tau} M \rightarrow \left[\tfrac{t}{x}\right] M \cup \bigcup_{x'\in(\tau\setminus t)} M \left[\tfrac{x'}{x}\right] M\ .$$

Finally, δ_q rules compliment N rules, by reducing the application of primitives to quantifiers in the opposite case, when all of the elements of the domain over which the quantifier is abstracted produce the same result under that primitive.

We must make one restriction to the form of the N rules. We are interested in lazy parameter-passing, and thus outermost reduction. But a rigid outermost application of N-rules would either not terminate, or would result in a separate case for each possible member of a type: hardly what one would consider appropriate! We will only consider a term to be an N redex, then, if the abstracted expression cannot be reduced (in this same qualified outermost manner) because the evaluation demands the contraction of a primitive applied to the abstracted variable corresponding to the abstraction, and this evaluation is not a δ_q redex. Thus, N-rules will only be applied when necessary.

Techniques for implementing β and δ reductions are well-known, and tag analysis techniques can easily be extended to implement σ and δ_q reduction. We are concerned here with the efficient implementation of N rules, in a way which maximizes the sharing across the alternatives created in

---

[2] `$dist` may be used to translate higher-level expressions such as

```
{x : x∈mySet, other_conditions}
```

into the core language as

```
$F x = ⟦{x : other_conditions}⟧
...($dist $F ⟦mySet⟧) ...
```

[3] *I.e.*, functions and sets are first-class language constructs; they may be passed to or returned by other functions and contained in other sets without restriction. Note that higher-order unification is *not* supported: higher-order unification is related to higher-order equality, which is not supported in pure functional languages.

N-rules, and describe an approach to their implementation below.

The rest of this paper is organized as follows: Section 2 describes the notion of *quasisharing*. Section 3 describes an extension to the G-machine which implements quasisharing. Finally, we discuss related and future work.

2 The Graph Model

In this section we present the way set-based functional-logic programs are to be represented in a graph. We will use this representation scheme in the following section in describing a graph reduction machine.

Functional Graph Reduction. The basic graph representation we use for program expressions is drawn directly from the standard graph representation of functional programs [Pey87, *e.g.*] and in fact coincides with such for purely functional expressions:

- Basic scalar values are tagged, for example,

  $\longrightarrow$

Int	24

  although in our presentation here we will omit the tag for the sake of brevity.

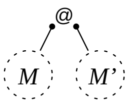
- References to combinator calls are tagged as such and include the arity of the function and the location of the code corresponding to that function:

  $\longrightarrow$

Fun	5	0xFF36

  When possible, we will simplify our figures here by showing only the name of the combinator in question.

- As usual, we represent curried function application using the '@' sign:



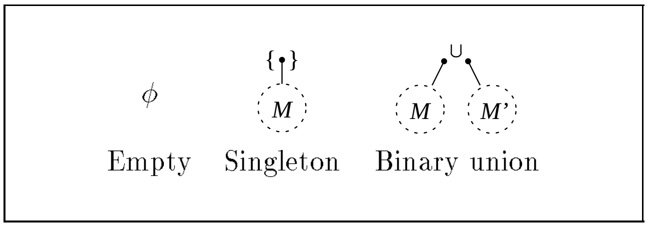
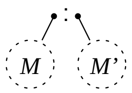


Figure 2: Graphs of basic set expressions

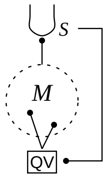
- Again to simplify the presentation, we consider only one data constructor, the familiar "cons" for pairs:



Quasisharing. We extend this standard graph model in two stages to include sets.

The first stage is simple: introducing constructions corresponding to the syntax of the set expressions themselves. The rendering of the empty set, singleton set and binary union are straightforward, and are shown in Figure 2.

We represent set abstractions in two graph items, one corresponding to the abstraction declaring the scope of the quantifier, and one corresponding to the quantifier itself: $\bigcup_{\mathtt{x} \in S} \mathtt{M[x]}$ will be drawn as



where QV is an abbreviation for quantified value. While these two graph entities are distinct—references to the former denote the value of the entire expression; those to the latter are made from within the abstraction, and denote the abstracted quantity itself—between them they store data relevant to the whole abstraction. The range of the quantifier and any constraints on values it may take are stored among these entities; in this presentation we indicate ranges with the abstraction node but a reverse arrangement is of course possible.

This first stage is insufficient by itself because of N-reduction. Let us assume that the expression

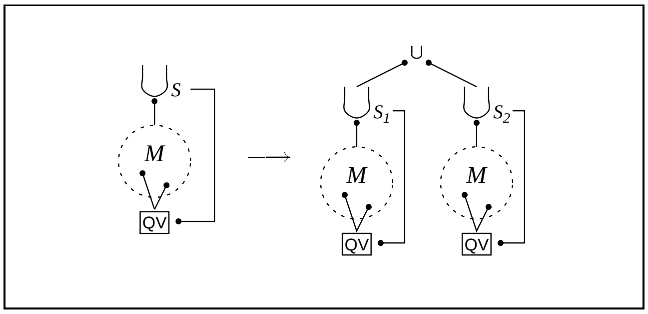


Figure 3: Naïve graph representation of the result of narrowing the range in the expression $\bigcup_{x\in S} M[x]$. The expression $M[\,]$ is duplicated once for each member of the resulting union.

$\bigcup_{\mathtt{x}\in S}$ `M[x]` from above is an N-redex,

$$\bigcup_{\mathtt{x}\in S} \mathtt{M[x]} \quad \longrightarrow_N \quad \bigcup_{\mathtt{x_1}\in S_1} \mathtt{M[x_1]} \ \cup \ \bigcup_{\mathtt{x_2}\in S_2} \mathtt{M[x_2]}$$

Applied naïvely, N-reduction would duplicate M, as shown in Figure 3.

In order to feasibly implement narrowing in graph reduction and avoid this duplication, we introduce a weakened form of sharing, which is what we call quasisharing. In the usual sense in functional languages, sharing is an all-or-nothing choice: either all of an expression is shared, or none of it is shared. We have no half-way point which allows specification of only a part of an expression as shared if the part which is not shared — from the view of the shared graph — corresponds to several different *sub*expressions. In other words (namely, those used by Lamping [Lam90]), the usual notion of sharing allows us to "fan-in" from several different expressions to a shared subexpression, but we have no way to "fan-out" from a shared expression to several different subexpressions. Since we cannot share the expression that results from narrowing, and since its wholesale duplication is not feasible, we will share those middle graph segments which are common among the different cases, and note where these shared areas end with distinguishable children.

The second stage of extending the functional graph representation to include sets introduces these markers which indicate such quasishared graph segments. An area of quasishared graph is marked by distinguished graph nodes called *quasisharing delimiters*, shown in Figure 4. These delimiters are given identifying tags in order to correctly associate corresponding *upper* and *lower* delimiters. This explicit association will be necessary in the case where narrowing over two different quantifiers occurs. The *arity* of a lower delimiter is the number of its children in the graph; all lower delimiters with the same identifying tag must have the same arity. The arity of an upper delimiter is taken to be the arity of the similarly tagged lower delimiters. Graph objects pointing to an upper delimiter may do so in a number of distinct ways equal to that delimiter's arity. Such distinct indication points specify corresponding child expressions at lower delimiters. A reference to the n^{th} box (from the left) of a labeled upper delimiter refers to the expression formed by using the n^{th} child (from the left) at lower delimiters of the same label. In Figure 5 we illustrate our notation for quasisharing and delimiters with the quasishared version of the same expression portrayed with naïve duplication in Figure 5.

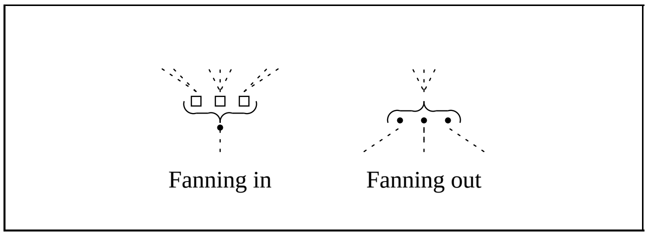


Figure 4: Upper and lower quasisharing delimiters.

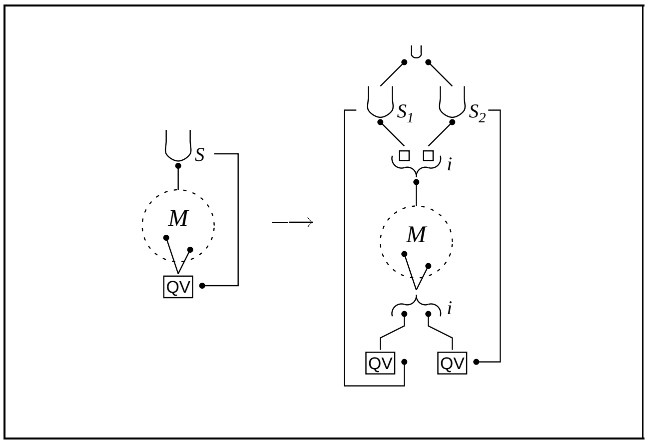


Figure 5: Quasisharing the result of narrowing.

Quasisharing and Optimal Reduction. Having described our technique, we can now more accurately characterize its relationship to Lamping's algorithm. On the surface, of course, is the difference in application: Lamping pursued an optimal reduction sequence for the (purely functional) lambda calculus; we are not concerned with any particular reduction strategy, but rather use a simplified technique to incorporate logic programming. Implicit in our presentation, and in agreement with more implementation-oriented discussions of graph reduction, is the directedness of graphs. Lamping's algorithm uses an undirected graph with explicit sharing: that is, while we allow arbitrarily many pointers to nodes, Lamping requires a "fan-in" node for *every* node which shares a subgraph. In the vocabulary of our presentation, one might say that sharing in Lamping's graphs is always quasisharing, and that when a node is fully shared there will simply be no corresponding diverging subexpressions. Lamping also deals with self-reference differently, by using explicit bracketing nodes to remove any possible ambiguities and to avoid duplicating the graph.

We did not incorporate these two mechanisms into quasisharing because of our emphasis on an abstract machine *which is useful for compilation*. It is not clear to us that the additional overhead required for undirected graphs and for bracketing will admit a good compilation, since the larger set of graph manipulation rules which results seems to be geared much more towards a graph interpreter than to a translation into a single fixed code sequence for each individual program. We do believe that quasisharing as we have presented it does lend itself well to compilation, and present a compilation scheme in the following section.

3 The Abstract Machine

In this section we sketch an extension to the G-machine which implements quasisharing. The G-machine was developed by Augustsson [Aug84] and Johnsson [Joh84] to implement lazy functional languages. Peyton Jones gives a thorough, easy to follow presentation of the G-machine on which much of our presentation here is based [Pey87].

Review of the G-Machine. The basic G-machine implements graph reduction of supercombinator definitions through code which first constructs the graph of program expressions, and then reduces the graph expression to normal form. Central to the operation of the G-machine is the notion of *unwinding application spines*. A *spine* of application nodes refers to the graph structure corresponding to a series of function applications. The expression $(f\ x_1\ \ldots\ x_n)$—that is, $(((f\ x_1)\ x_2)\ \ldots\ x_n)$—is represented by the graph shown in Figure 6. Repeated *unwinding* of spines is the process by

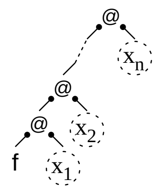


Figure 6: An application spine.

which the graph expression is reduced to a value. A spine is unwound beginning from its "top" (outermost) application node, and down the chain of left-side application nodes until a function definition is found. This function will direct the construction of new graph elements (which may of course re-use the existing values) to replace all or part of the spine. Unwinding is repeated until a head-normal node results.

The G-machine itself consists of four[4] components: (1) a *code sequence*, consisting of G-code instructions to be executed, (2) a *heap* mapping addresses to graph units, where the program graph is stored, (3) a *stack* of addresses and (4) the *dump*, a second stack of pairs (closures) of a code sequence and an address stack. We will describe the operation of these structures in sufficient detail to later illustrate the modifications for quasisharing.

The stack is used in two different ways, one during unwinding and one during actual function execution. During unwinding, the spine nodes themselves are placed on the stack as the spine is unwound (Figure 7). Upon finding a function, one of

[4] The original G-machine papers included additional

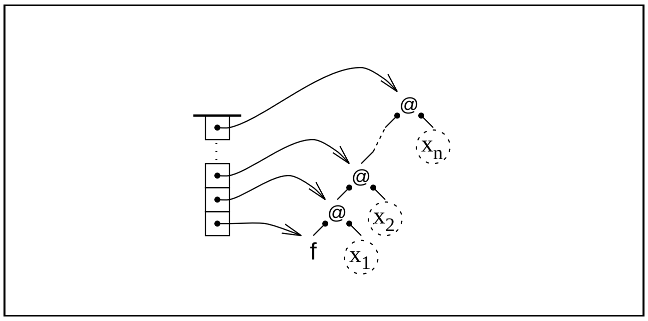


Figure 7: Using the stack to unwind a spine. We portray the stack as growing *downward.*

two alternatives may be taken. If the function is applied to fewer arguments than its arity, the expression at the top of the spine denotes a function, and is thus a weak head-normal form value.

Otherwise, the stack is reconfigured for the execution of the function. Let us assume the general case that the function f is of arity i, $i < n$: that is, the return value of f should be a function, which will then be applied to the remaining $(n - i)$ arguments. The address of the outermost application node of the subexpression $(f\ x_1\ \ldots\ x_i)$ is kept on this stack, and the i arguments are also placed on the stack (rather than the application nodes which point to those arguments): the function will (likely) need to access those arguments in building the result, and will need to *update* the graph with that result. The remainder of the old stack—that part holding the spine of the application of the other $(n - i)$ arguments—is stored in the dump, which we describe in more detail below. Figure 8 shows this division of the stack. The G-code sequence for a function should then (1) construct the graph of its result from the arguments on the stack, (2) update the root of the outermost application with this result and (3) return to another unwinding of this result (since the result may be yet another function application; if the result is known at compile-time to be a value than the function's code may return directly to the top closure on the dump).

The dump is used to store partially-performed reductions which must be temporarily suspended until some other reduction is carried out. For example, an `if`-expression cannot be contracted until its boolean predicate is evaluated; if the reduction

components.

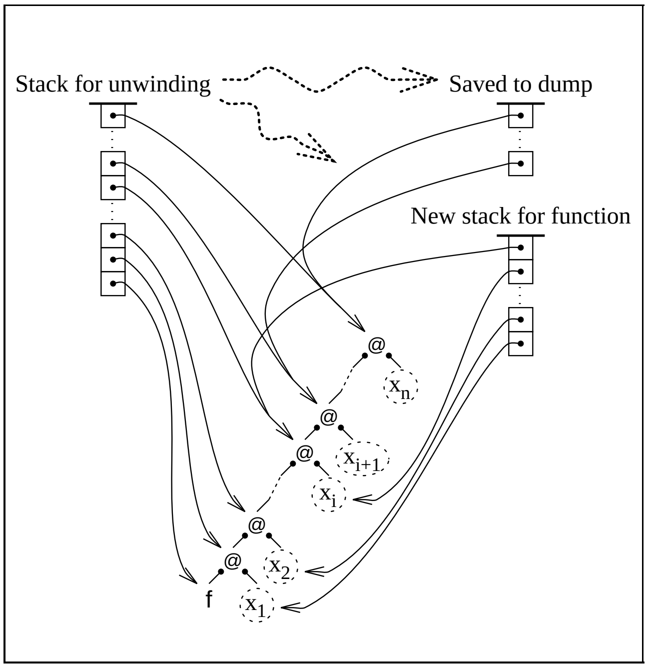


Figure 8: Reconfiguring the stack, from unwinding to function execution.

of this argument requires unwinding then we must store the reduction of the `if`-expression so that it may be resumed later. Similarly, we must store the partial spine holding the extra $(n - i)$ arguments in Figure 8; f will return a function to use these arguments, but in order to get that function we must use the stack.

When the reduction of some subexpression is completed, say with stack S, the first stored code sequence and stack on the dump are then restored as the current code sequence and stack, with the address at the base of S pushed onto the top of the newly restored stack,

$$\begin{array}{lccc}
\underline{\text{Code}} & C_0 & \longrightarrow & C_1 \\
\underline{\text{Stack}} & a_0 : \ldots : a_n & \longrightarrow & a_n : S_1 \\
\underline{\text{Dump}} & \boxed{\begin{array}{c}\langle C_1, S_1\rangle \\ \langle C_2, S_2\rangle \\ \vdots \\ \langle C_m, S_m\rangle\end{array}} & \longrightarrow & \boxed{\begin{array}{c}\langle C_2, S_2\rangle \\ \vdots \\ \langle C_m, S_m\rangle\end{array}}
\end{array}$$

thus proceeding with whatever manipulation was being performed to the graph on the newly restored stack S_1. Continuing the examples of the previous paragraph, having reduced the first argument of an `if`-expression to a simple boolean value, we would then test that value; having computed a function value to be applied to further arguments, we would continue unwinding the original spine. We illustrate this latter case in Figure 9. If $(f\ x_1\ \dots\ x_m)$ reduces to

$$f'\ x'_1\ \dots\ x'_p$$

where the arity of f' is m', $p < m'$, then the unwinding which had spawned the reduction of $(f\ x_1\ \dots\ x_m)$ in the first place is resumed with the root node of $(f'\ x'_1\ \dots\ x'_p)$.

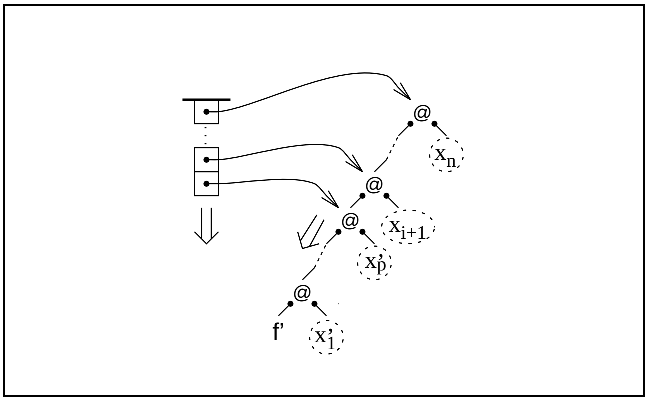


Figure 9: Unwinding should proceed in the direction of the arrows $\Longrightarrow$.

At the highest level, the G-machine is directed by an *evaluate/print* loop: graph expressions are reduced until the outermost node of their graph is that of an expression in weak head-normal form; this node is then printed, and any child nodes it may have are then (in the loop) evaluated and printed.

The description of the G-machine we have given here is not complete; in particular we have omitted any description of the G-code instruction set. Rather than provide such a comprehensive reference, we have outlined the fundamental operations of a basic G-machine which are relevant to our adaptation of the machine for quasisharing. We have also omitted any discussion of the numerous optimizations to the G-machine, some of which introduce great variations of this scheme; we shall return to this subject later.

Sketch of the NG-machine. In this section we outline the extensions to the G-machine mechanism needed to implement quasisharing. The common theme of these extensions is to broaden the use of

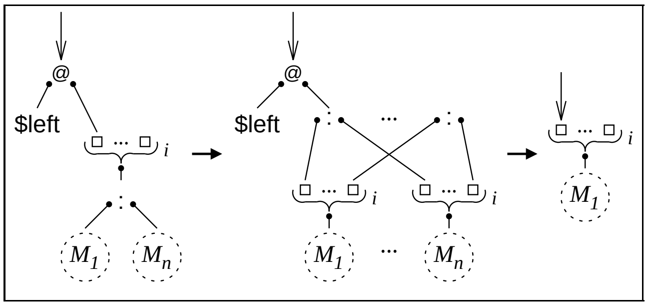


Figure 10: Unsharing a pair node in order to export its subgraphs.

the dump to store closures involving graph structures other than just application spines. These structures can then be used as a sort of marker to allow the propagation of delimiters described in §2. We have named the new abstract machine NG, for Narrowing G-machine.

While evaluations performed by the NG-machine shall of course adhere to the standard reduction outlined in §1, we have not yet described any standardized scheme for manipulating quasisharing delimiters. The abstract machine shall treat these structures as follows:

1. Upper delimiters shall be propagated downward only when it is necessary to access some child graph of the node immediately below that delimiter.

   For example, consider a pair node below an upper delimiter: Before the left child M_1 of the pair node may be exported as the pair's left element (for example), the pair node must be unshared, and thus duplicated, so that the child nodes may be treated as properly quasishared, as shown in Figure 10.

2. Unlike our treatment of upper delimiters, we shall propagate lower delimiters upon any call to evaluate them. The motivation behind such a propagation follows from two facts:

   (a) We will only ever evaluate the graph of an expression—here, the lower delimiter—if it is absolutely necessary to do so.

   (b) It is reasonable to expect that distinctly-valued expressions beneath the lower delimiter will produce distinct values when used in other expressions.

That is, when evaluating an expression we know that the result will be used somewhere else, and so we perform the necessary unsharing prior to reduction, and rid ourselves of having to cope with delimiters during the actual reduction.

Moreover, we know that the "somewhere else" which requires evaluation of the expression under the delimiter is itself required somewhere else again, and so on all the way to the corresponding upper delimiter. Therefore, we will propagate lower delimiters upwards, not just once, but all the way to that matching upper delimiter.

Implementing the first propagation is straightforward: abstract machine instructions such as `Left` which export substructures must insure that upper delimiters first be moved downward.

Implementing the second propagation uses the dump. When we cause the evaluation of the expression under an upper delimiters—simply an evaluation, not a deconstruction, which would necessarily be preceeded by a evaluation anyway—we store the delmiter on the dump. If we subsequently find a matching lower delimiter, we unshare the structures on the dump between the two matching delimiters: each closure on the dump is precisely the "somewhere else" which requires the evaluation of the next closure to the top! Thus we have propagated the lower delimiter all the way to the upper delimiter, and can remove both and then proceed with reduction. Figure 11 shows the general dump/stack configuration corresponding to the situation where we perform such a propagation.

Building a mechanism to introduce delimiters into the graph in the first place requires another type of node on the stack: the quantifier nodes. In fact, the overall mechanism includes the propagation described above. When evaluating the expression beneath a quantifier node, we may of course discover that the quantifer is in fact an N-redex, which requires the introduction of delimiters. But the reasoning above which calls for unsharing the spines between matching delimiters also calls for unsharing here, since the lower delimiter thus introduced will immediately be required just as above. By storing the quantifer on the dump we can again know exactly what graph nodes must be unshared and duplicated.

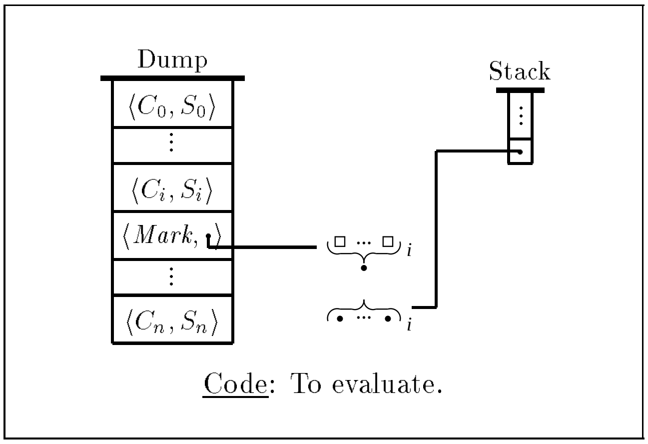


Figure 11: Dump/stack configuration requiring upper propagation of a lower delimiter.

Disequality between quantifiers can be treated in much the same way. We record finding disequality information within the graph on the dump, and then upon comparing two quantifiers for equality we initiate a search in the dump for that information. The search ceases when either such information, or the abstraction node (and hence the limit of the scope of) one of the quantifiers is found. In the former case the comparison can be immediately concluded, as the two are known unequal; in the latter case, the abstraction itself forms an N-redex, and so we must split the set and introduce quasisharing.

Again, we have given only a sketch of the abstract machine; a complete description of the machine states is quite lengthy and can be found in [Mar93].

4 Related Work

Mück [Müc92] wanted an abstract machine for which formal verification would be possible, so he based his approach on the Categorical Abstract Machine (CAM) [Cou90]. The CAM executes categorical combinators directly as machine instructions, providing a compromise between abstraction and efficiency. Mück described the CAMEL, an extention of the CAM with logic programming features, *viz.*, a stack of logical variables (quantified values), unification instructions and a backtracking mechanism. In contrast with the CAMEL, our approach does not provide unification as an atomic operation, but rather breaks it down into smaller units interspersed with other reduction operations. The backtracking mechanism in the CAMEL implements OR-parallelism; our machine represents the analogous operation (set union) by copying or sharing graph components. With the latter approach true parallel execution would seem to be more feasible. The CAMEL implementation is eager, though Mück suggested that the methods for making the original CAM lazy would apply to the CAMEL as well. Cousineau's modification to the CAM instruction set for lazy evaluation involves applying a substitution, replacing occurrences of a closure by its computed result to every expression in the execution stack. This technique is useful for its clarity, but is certainly not suited to an efficient implementation.

The Babel Abstract Machine (BAM) is described as a graph machine, in that activation records are joined into circular linked structures in the heap (rather than, say, a simple stack). The first implementation [KLMR90] was eager, but was later adapted to a sort of lazy evaluation [MNKLRA90]: the adaptation does introduce an almost-outermost evaluation order, but does not permit the structure sharing found in traditional graph reduction systems. As in the CAMEL, OR-parallelism is implemented via backtracking; the BAM maintains stacks of logical variable bindings and backtracking points.

Reddy described an operational semantics for functional logic programming in which the various occurrences arising from multiple uses of an abstraction parameter in a function body within a single nondeterministic choice branch are shared, but does not provide for sharing across nondeterministic choice brances (i.e., across alternative narrowings) [Red86].

In his thesis [Loc92], Lock creates an implementation language suitable for describing a wide variety of functional logic languages. This language is essentially a λ-calculus enriched with operations for choice, unification and annotations from controlling the evaluation strategy. He then presents an abstract machine, the JUMP machine, onto which this implementation language can be compiled. This abstract machine is essentially an orthogonal combination of the Spineless Tagless G-Machine [Pey92] with the Warren Abstract Machine [War83]. This machine integrates three basic machine concepts: the conventional stack-based architecture that ensures efficient expression evaluation, the choice point technique that implements backtracking, and the closure technique that provides a uniform representation for all kinds of functional and logical machine objects. It automatically detects deterministic computations at run-time (expressions which do not contain logic variables) so that purely functional execution is not slowed by the logic programming capability. To insure that non-strict evaluation will not cause re-evaluation of arguments, compilation of the source language into his enriched λ-calculus identifies multiple references to a function argument, and generates code which implements sharing via indirection nodes. With respect to sharing across narrowings, Lock notes that the evaluation of a subexpression that does not reference narrowed logical variables within its scope could in principle be shared across the various narrowing. However, it is an open problem as to whether such expressions can be detected at compile time: since Lock's system generates instructions for sharing at that time, no sharing is performed across narrowings. On the other hand, our system detects opportunities for sharing across narrowings at run-time[5].

Liu [Liu93] presents a graph reduction scheme which, unlike the others, does allow some sharing across or-parallel branches. He draws mainly from techniques for AND/OR tree rewriting. Under his scheme, references to an expression may share a single copy of the corresponding graph across OR-parallel branches. Simple reductions within that expression — reductions which do not bind shared logic variables — may update the shared graph. As in our work, Liu's motivation is that, even where copying is inevitable, reductions which would be common to all the copies are best done before copy-

[5] More precisely, our system assumes that expressions may be so shared until, at runtime, it discovers otherwise.

ing. When a shared logic variable within such a shared graph must be bound, the AND/OR tree corresponding to the *entire* shared expression is duplicated. Our technique takes sharing across narrowings one step further, in that such duplication is performed lazily, duplicating only those subterms of the shared expression which are immediately affected by the values of the narrowed variables. Other subterms of the once-shared expression remain shared across all the newly-formed copies, as *their* initial reductions might not depend on the narrowing chosen for the sake of a parent node. Liu's graph rewriting model is based on concurrent processing of an AND/OR tree. Process tokens represent processes traversing and rewriting the tree; these contain four elements: a process label to index variables in the clause and to mark the origin of a shared process, a local environment to store local bindings from the head unification between a goal and a clause, a binding array to contain global bindings which will contribute to a computed answer answer substitution, and a process origin to record the context in which the shared process was started. Our system should avoid some of the overhead associated with Liu's system of concurrent processes, as the order of operations is graph driven (and so does not rely on concurrent-processing control structures).

With the exception of Liu's scheme, the implementations described above unbind logical variables upon backtracking, discarding all changes to the environment and new allocations made since the binding, and using the restored, original environment in the computation of alternate narrowings. Rather than considering alternatives separately, and arbitrarily discarding all such changes, our approach allows the same context to be shared between all alternatives simultaneously, creating distinct subexpressions only as needed, and without discarding potentially useful results — those not directly dependant on the specific binding — from other alternatives. Further research will determine which approach produces the greater savings overall.

While all of the approaches described here, excluding Liu's model but including ours, are designed for single-processor targets, our approach seems better suited for use as a model for compilation to parallel architectures, at least under a shared-memory environment (Liu's model was designed with parallel targets in mind). In backtracking-based approaches, solutions must be computed in sequence; this approach seems to require extensive revision to describe a parallel search for multiple solutions. In our abstraction the computation of one solution never requires the dismantling of data structures that were used in computing another; where a quasishared area of the graph is accessible to all processors reducing alternatives referring to that quasishared graph, the adaptation to a parallel target seems straightforward.

5 Conclusion

The notion of quasisharing is an interesting technique to increase the degree of sharing in implementations of functional-logic languages. We are currently devising an implementation based on the modified G-machine described here, which will help to judge the benefit one can actually achieve.

Several interesting questions remain open. First, it would be interesting to determine the degree to which quasisharing can be implemented in existing implementations. Much work has been done on efficient but non-quasisharing implementations, and it would certainly be wise to take advantage of those technologies. A second open question concerns the adaptation of existing G-machine optimizations to the NG-machine outlined here. A worthwhile implementation of the NG-machine would require — as do worthwhile implementations of the G-machine — significant optimization to the reduction mechanism. However, many existing G-machine implementations revolve around preventing the allocation of short-lived heap nodes corresponding (among other things) to spine nodes which are immediately re-unwound and discarded. However, the quasisharing technique explicity uses spines: adaptation of such optimizations would require a more spineless manipulation of quasisharing. Finally, the NG-machine forms an operational semantics for the source language, and a more rigorous examination of this semantics would be interesting.

References

[Aug84] Lennart Augustsson. A compiler for lazy ML.

In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, August 1984.

[Cou90] G. Cousineau. The categorical abstract machine. In G. Huet, editor, *Logical Foundations of Functional Programming*. Addison Wesley, 1990.

[DFP86] John Darlington, A. J. Field, and Helen Pull. Unification of functional and logic languages. In DeGroot and Lindstrom, editors, *Logic Programming, Relations, Functions and Equations*, pages 37–70. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedins of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, June 1984.

[KLMR90] Herbert Kuchen, Rita Loogen, Juan José Moreno-Navarro, and Mario Rodríguez-Artalejo. Graph-based implementation of a functional logic language. In *Lecture Notes in Computer Science 432*. Third European Symposium on Programming, May 1990, Copenhagen, Denmark, Springer-Verlag, 1990.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Conference Record, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*. ACM Press, January 1990.

[Liu93] Feixiong Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *Sixth ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 201–209. ACM Press, June 1993.

[Loc92] Hendrik C. R. Lock. *The Implementation of Functional Logic Programming Languages*. PhD thesis, GMD, Universität Karlsruhe, 1992.

[Müc92] Andy Mück. CAMEL: an extension of the categorical abstract machine to compile functional/logic programs. In *Lecture Notes in Computer Science 631*. Programming Language Implementation and Logic Programming, August 1992, Leuven, Belgium, 1992.

[Mar93] John Maraist. An abstract machine for compiling set-based functional logic languages. Master's thesis, Tulane University, New Orleans, Louisiana 70118, August 1993. Revised version, November 1993.

[MNKLRA90] Juan José Moreno-Navarro, Herbert Kuchen, Rita Loogen, and Mario Rodiguez-Artalejo. Lazy narrowing in a graph machine. In *Lecture Notes in Computer Science 463*, pages 298–317. ALP 2nd International Conference, Nancy, France, 1990.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[Red86] Uday S. Reddy. *Logic Languages Based on Functions: Semantics and Implementation*. PhD thesis, University of Utah, 1986.

[SJ92] Frank S. K. Silbermann and Bharat Jayaraman. A domain-theoretic approach to functional and logic programming. *Journal of Functional Programming*, 2(3):273–321, 1992.

[War83] D. Warren. An abstract instruction set for prolog. Technical Report 309, SRI International, 1983.