A Graph Reduction Technique with Sharing across Narrowings for Functional-Logic Languages*

John Maraist† Frank S. K. Silbermann‡

September 22, 1994

Presented at PLILP'94, the Sixth International Symposium on Programming Language Implementation and Logic Programming, Madrid, Spain, September 14-16, 1994. Appears in the conference proceedings, Lecture Notes in Computer Science 844, p. 355-369. This version of the paper is set in larger type and with different margins to allow larger (and hopefully, more readable) graphics than were possible in the conference proceedings, and contains corrections to some typographical errors, but is otherwise the same as in the conference proceedings.

Abstract

We present a system for the efficient implementation via graph reduction of functional-logic programming languages. In contrast to existing systems, ours allows sharing not only of fully common subexpressions but also of partially common expressions formed across narrowings, and allows both lazy and eager evaluation semantics. This sharing across narrowings is achieved by adapting a technique described by Lamping for optimal reduction of lambda terms.

Implementations of functional-logic languages, by and large, use some form of backtracking when rebinding logical quantifiers after the violation of some restriction by a previous binding. Such backtracking will usually discard values computed since the original binding, but often some portion of those values will not be affected by the rebinding, and so must be recomputed wastefully. In this paper we present a *set-based* graph reduction technique for a lambda-calculus extended by relative set abstraction which allows such subexpressions to be shared across alternative bindings to quantifiers. Our technique is similar to an algorithm of Lamping for optimal reduction of lambda terms [5]; we defer a more detailed comparison to the conclusion of this paper.

Terms and Graph Expressions

Like Lamping, we will use a graphic rendering of the terms of our calculus, rather than a textual rendering, due to the non-linear nature of the layout of terms under graph reduction. Our source

*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 und Datenorganisation (IPD), Universität Karlsruhe, 76128 Karlsruhe, Germany. Email: `maraist@ira.uka.de`.

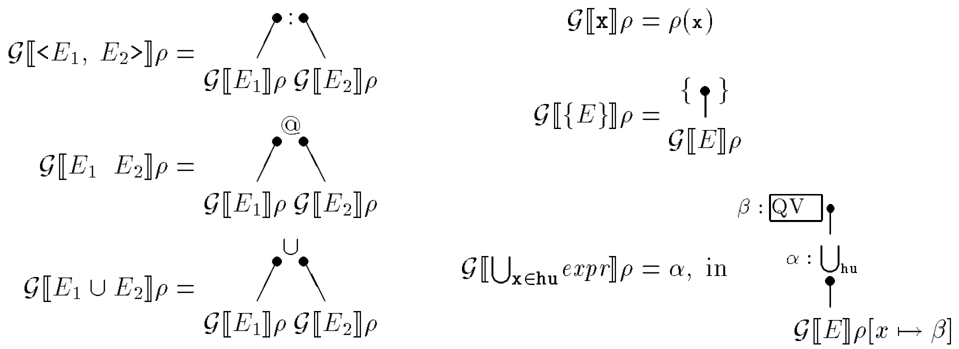
‡Postal address: Department of Computer Science, 301 Stanley Thomas Hall, Tulane University, New Orleans, LA 70118, USA. Email: `fs@cs.tulane.edu`.

language is of course textual, and is derived from Silbermann and Jayaraman's syntax [11]. We take a program to be a finite number of *supercombinator* definitions, that is, a binding of a name (which by convention is prefixed by a '\$') to a closed term; the term may itself denote a function value, but is allowed to contain lambda expressions only at the top level. This restriction to supercombinator form is not a serious limitation; several algorithms exist to convert arbitrary lambda expressions into supercombinator form [9, e.g.]. Formally, supercombinator definitions match the following grammar:

$$
\begin{array}{lll}
\mathit{defn} & \texttt{::=} & \texttt{\$}\mathit{name}\ x_1\ \ldots x_n\ \texttt{=}\ \mathit{body} \\
\mathit{body} & \texttt{::=} & \bigcup_{\texttt{x}\in\texttt{hu}}\ \mathit{expr} \\
 & \texttt{|} & \mathit{expr}
\end{array}
\qquad
\begin{array}{lll}
\mathit{expr} & \texttt{::=} & \mathit{atom}\ \texttt{|}\ \mathit{bool}\ \texttt{|}\ \texttt{\$}\mathit{name}\ \texttt{|}\ x \\
 & \texttt{|} & \texttt{<}\mathit{expr}\texttt{,}\ \mathit{expr}\texttt{>}\ \texttt{|}\ \mathit{expr}\ \mathit{expr} \\
 & \texttt{|} & \phi\ \texttt{|}\ \{\mathit{expr}\}\ \texttt{|}\ \mathit{expr} \cup \mathit{expr}
\end{array}
$$

We reserve certain supercombinator names for primitive operations: `$left` and `$right` for projection of pair elements; `$not`, `$and` and `$if` for boolean values; `$atom?`, `$bool?`, `$pair?`, `$func?` and `$set?` for dynamic type checking (or alternatively, one might assume that programs consist only of well-typed expressions); `$equal`, over first-order terms; `$dist`, for set derivations, which we describe below; and for other primitive operations over atomic elements. Moreover, we shall expect the name `$main` to be defined in all programs and to take no arguments; the body of `$main` is taken to be the value of the program.

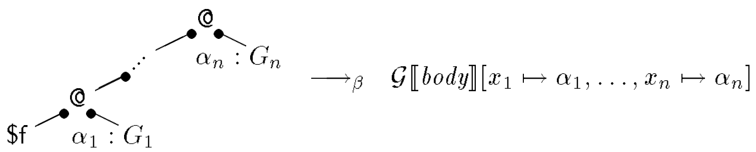
We map textual expressions which are closed with respect to some environment to graph expressions with $\mathcal{G}$. The graph representation of simple values, supercombinator names and the empty set are simply the values and names themselves as childless graph nodes. For larger expressions, $\mathcal{G}$ is defined as follows:



The environment contains references to other graph elements.

Reduction of Graph Terms

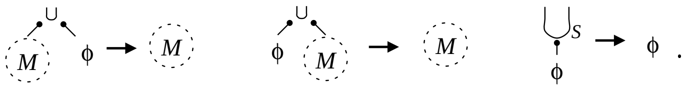
We identify five notions of reduction over graph terms. Two of these are well-known: β and δ are standard notions of reduction for lambda expressions [1], for the application of lambda abstractions and of primitive functions, respectively. Here, β-reduction take the application of a supercombinator to enough arguments into the body of the supercombinator with the formal parameters instantiated: for `$f` $x_1\ \ldots x_n\ =\ \mathit{body}$,



and similarly for δ over primitive functions applied to elements of their domain.

The other two notions of reduction concern the set constructs with which we implement logic programming. First, σ-reductions manipulate graphs in the way set constructs should interact with each other:

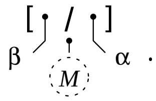
1. The union of any expression with the null set term reduces to that expression again:



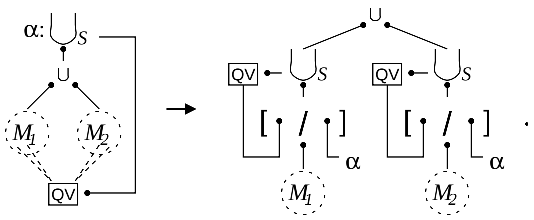
2. Set abstraction should be distributed over binary unions. Textually, we have

$$\bigcup_{x\in S}(A \cup B) \longrightarrow (\bigcup_{x_1\in S}[x_1/x]\,A) \cup (\bigcup_{x_2\in S}[x_2/x]\,B)\ .$$

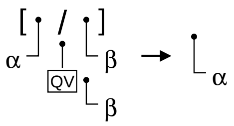
   Substitution must be explicitly performed, and we introduce a new graph node for this operation. For an "old" abstraction at location α, a "new" quantified value at location β and an expression M we have the graph



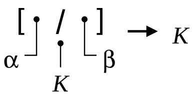
   So the graph manipulation for the distribution of an abstraction over a binary union is then



   and the substitution node interacts with the other graph nodes as follows:

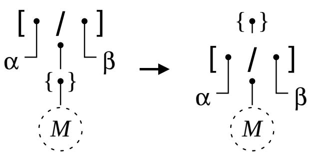


(a)

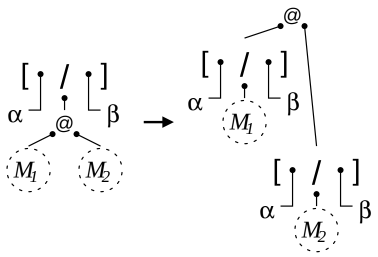


(where K is an atomic or boolean constant, supercombinator reference, the null set node or a quantifier whose corresponding abstraction is *not* labeled β)

(b)



(c)



(d)

Substitutions distribute over pair and union constructors in the same way as for application nodes.

3. We can derive one set from another with the $dist operator. Applications of $dist reduce as follows[1]:

```
$dist φ f     = φ
$dist {x} f   = f x
$dist (x∪y) f = ($dist x f) ∪ ($dist y f)
$dist (⋃x) f  = ⋃($dist x f) .
```

Here `f` is a function from individual set elements to sets; the result of a call to `$dist` need not be isomorphic to its set argument. This `$dist` function is useful as a low-level view of abstraction over a user-defined set (operationally, abstraction over the Herbrand universe is treated as a special case). To translate a higher-level language expression like

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

into the core language, we would write

$$\texttt{\$F x = } [\![\texttt{\{x: other\_conditions\}}]\!]_{L\rightarrow core}$$
$$\texttt{\$expr = \$dist \$F } [\![\texttt{mySet}]\!]_{L\rightarrow core} \ .$$

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 [9, 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 [2] without imposing any inelegant first-order restrictions on purely functional expressions. N-rules are derived from equations partitioning sets in meaningful ways. In the untyped calculus we illustrate here, we denote the collection of first-order terms by `hu`, for the Herbrand universe, and allow the programmer to abstract over these terms. To reduce the application of (say) the $atom? primitive to a quantifier abstracted over `hu`, we must use the relationship between `hu` and the collection of atomic elements,

$$\mathsf{atoms} \cup \mathsf{bools} \cup \textstyle\bigcup_{x\in\mathsf{hu}} \bigcup_{x\in\mathsf{hu}} \langle x,\ y\rangle \quad ,$$

to derive a rule for reducing abstractions over `hu`:

$$\bigcup_{x\in\mathbf{hu}} M_x \quad \longrightarrow_N \quad \bigcup_{x_1\in\mathbf{atoms}} M_{x_1} \ \cup \ \bigcup_{x_2\in\mathbf{bools}} M_{x_2} \ \cup \ \bigcup_{x_3\in\mathbf{hu}} \bigcup_{x_4\in\mathbf{hu}} M_{\langle x_3,x_4\rangle} \ . \tag{1}$$

Similarly, we can narrow the base of abstractions in other ways with rules derived from other equations:

$$\mathsf{hu} = \{n\} \cup \textstyle\bigcup_{x\ \in \mathsf{hu}\setminus\{n\}}\{x\}, \text{ for } n \in \mathsf{hu}$$
$$\mathsf{bools} = \{\mathsf{true}\} \cup \{\mathsf{false}\}$$
$$\mathsf{atoms} = \{a\} \cup \textstyle\bigcup_{x\ \in\ \mathsf{atoms}\setminus\{a\}}\{x\}, \text{ for } a \in \mathsf{atoms}$$

[1] We abuse the notation here, and write graph rules textually, to save space.

One would also expect N-rules based on individual δ rules, for narrowing required by primitive functions applied to quantifers abstracting over sets whose elements do not all produce the same element under the primitive in question. In a typed calculus, where one would abstract over an algebraic type rather than the (untypable) collection of all first-order terms, N rules would also reduce an abstraction over a sum type to the union of abstractions over the component types of the individual tagged summands.

Finally, δ_q rules contract applications of primitive functions to quantifiers where N rules do not apply. One must not necessarily reduce an abstraction containing such an application to a union of smaller applications; when every item in the set *does* produce the same result, contraction can simply produce that result. For example, a δ_q rule will allow us to contract:

$$\bigcup_{x\ \in \mathsf{atoms}} \{\texttt{\$atom?}\ x\} \rightarrow_{\delta_q} \bigcup_{x\ \in \mathsf{atoms}} \{\texttt{true}\}\ .$$

Techniques for implementing β- and δ-rules in graph reduction systems are well-known [9]. Existing techniques for pattern matching on structures, such as using the address of a jump table as the tag of graph nodes [9, §19.4.1, p. 330-1], may be easily adapted for σ-rules, and δ_q-rules are a simple extension of ordinary δ-rules. Efficient implementation of the N-rules in a way which maximizes sharing among the resulting expressions is more difficult. The main contribution of this paper is the notion of *quasisharing*, which allows such sharing, and which we describe below.

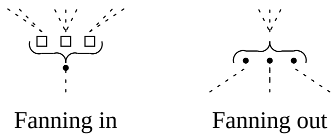
Reduction Strategy. We assume the existance of some notion of evaluation context [3] which allows us to select the next redex to be contracted. Since the binary set disjunctions are in a sense non-deterministic, and since any complete evaluation strategy will involve some parallelism [11], we would not necessarily expect that a unique redex should be chosen at every step, although an actual implementation might reasonably make such a restriction. We are primarily interested in lazy evaluation here, so for reduction to weak head-normal form we will take evaluation contexts to be terms matching E in the grammar

$$E \ ::= \ [\,] \mid E\ T \mid E \cup T \mid T \cup E \mid \{E\} \mid \textstyle\bigcup_{x \in S} E\ .$$

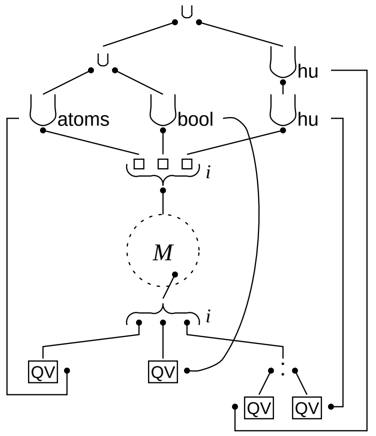
Complete reduction within structures or of abstractions will require a more elaborate scheme for evaluation contexts, but this scheme will suffice for our presentation here. Although we do concentrate on an outermost reduction scheme, our technique is not tied to any particular strategy, and it would be straightforward to adopt our technique to (say) a call-by-value framework.

Quasisharing

A direct implementation of the N rules as stated would be unthinkable, as the rules describe large-scale duplication of graph structures. Existing functional-logic implementations resolve this difficulty by generating only one binding at a time, constructing alternative branches only upon a failure in the first branch. But then opportunities for sharing across narrowings are lost, as reductions between rebindings discard values and expressions not needed under each binding in turn. In order to feasibly implement N rules without losing this sharing across alternative bindings, 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 [5]), 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 distinct children. It is this middle section of graph to which we refer as quasishared. An area of quasishared graph is marked by distinguished graph nodes called *quasisharing delimiters*:



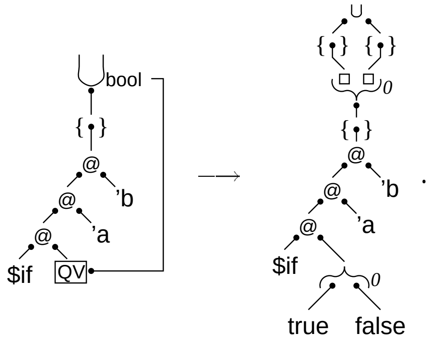
These delimiters are given identifying tags in order to correctly associate corresponding *upper* and *lower* delimiters. This explicit association will be necessary when narrowing over two different quantifiers occurs. The *arity* of a lower delimiter is the number of its children in the graph; all 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. For example, the application of rule (1) to the term $\bigcup_{x \in \mathbf{hu}} M[x]$ would produce the expression



.

Each N-rule is then implemented by the introduction of a quasishared disjunction. However, one important restriction on the application of these rules must be observed: that within the bodies of the abstractions narrowing produces, we must be able to contract expressions which could not be reduced in the body of the original abstraction prior to narrowing. This condition will help to avoid dividing sets unnecessarily. Unfortunately, it will not always be immediately apparant whether the N-rule will create such new redexes; as we discuss below, some propagation of the delimiters may be required before redexes may be contracted. We state the condition in terms of evaluation contexts: rather than any term of the form $\bigcup_{x \in S} M$, a term will only be taken as an N-redex if it is of the form $\bigcup_{x \in S} E[\$\mathsf{f}\ x]$, and moreover, if $(\nexists! y)(\forall x \in S)\, f \twoheadrightarrow y$ as well.

We illustrate the introduction of quasisharing delimiters, and motivate how these delimiters should be propagated through the graph, with a simple example program[2], `$main =` $\bigcup_{\mathtt{x} \in \mathtt{bool}}$`{$if x 'a 'b}`. Translating into a graph term and reducing, we have

$\mathcal{G}[\![$`$if x 'a 'b`$]\!][\,] =$ 

Unfortunately, the resulting graph expression is also not immediately reducible. Narrowing was performed to resolve a situation where different possible groundings of a quantifier would produce different results of a call to a primitive function: namely, a call to `$if` on the two different boolean values. Given such a situation, we can expect to find that some items which are quasishared right after narrowing will have to be unshared later. Of course, for this example the unsharing is trivial. The spine must be duplicated and unshared, but the other ribs may be treated in any of several ways; they may be fully shared, they may remain quasishared (by inserting new upper delimiters above them) or else they may be duplicated: here, the expressions at the two other ribs are so small, it really does not matter which option we choose.

Propagating Delimiters. We consider a second sample program to further illustrate how delimiters should move through the graph:

```
$f x = $if ($bool? x) ($g x) ($h x 'a)
$g x = $if x ($h x 'b) ($k x)
$h x y = {<x, M_y>}
$k x = {x}
$main = ⋃_{x∈hu} ($f x)
```

Here M_y is a (large) expression in which the formal parameter `x` does not appear. The first part of this reduction is shown in Fig. 1. Element 1.a of the figure shows the initial graph of the program, and the unfolding of the call to `$f` is in 1.b. For further reduction, we must evaluate the call to `$bool?` in order to produce a proper boolean value for `$if`. Since the Herbrand universe contains some elements which are boolean values and others which are not, we must separate the booleans from the non-booleans (element 1.c). But to allow the call to `$bool?` to be reduced, the lower delimiter must be moved upwards and the spine duplicated, since we expect that not all of the narrowed cases will produce the same value (if they *would*, then narrowing would have been unnecessary in the first place). After that propagation (element 1.d), `$bool?` is still being applied to non-grounded quantifiers, but the applications can all still be reduced, since for each individual quantifier, `$bool?` maps all allowed values to the same result. namely, boolean values will return `true`, while atoms and pairs will not (element 1.e).

Once again a lower delimiter must be propagated upwards, and graph elements duplicated (element 1.f), in order to allow reduction to continue: `$if` will of course behave differently for

[2]Strictly speaking, this program is not in our grammar. However, as we shall see, its graph could easily be the intermediate result of a larger program.

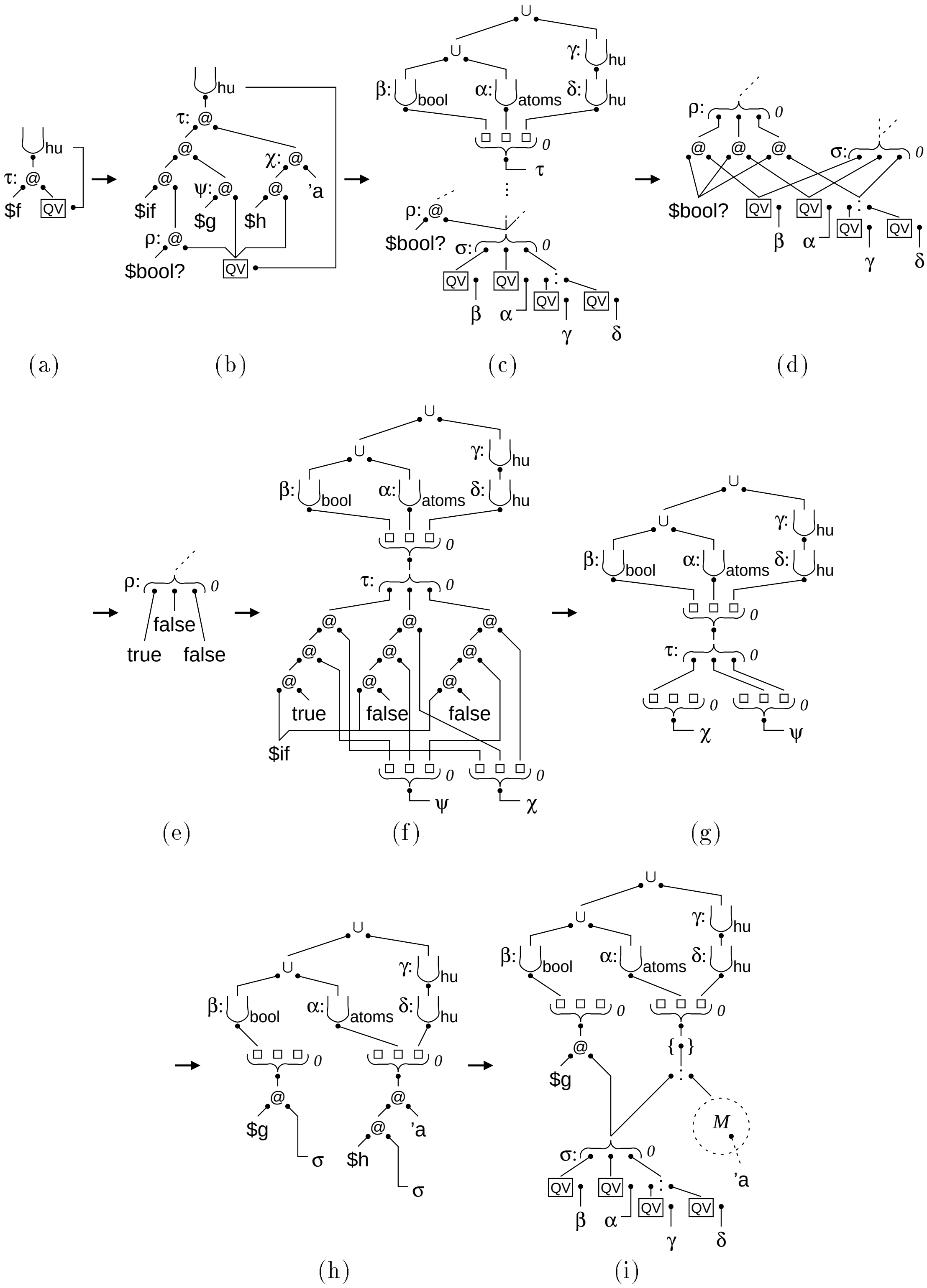


Figure 1: First steps in the reduction of Ex. 2.

the different boolean values. The spine for a call to $if is different from the spine for the call to $bool? in that the former spine contains other ribs besides the one from which a lower delimiter must be progated up. By dealing carefully with such cases, we can use quasisharing to save work. We will not necessarily have to duplicate these other ribs; in fact the only reason any of them might have to be duplicated would be if their evaluation entailed the same sort of application-applied-to-lower-delimiter situation which has already forced the duplication of the spines for the calls to $bool? and $if. Therefore, rather than duplicating them we place a "new" upper quasisharing delimiter, given the same identifier as the lower delimiter being moved up the spine, at the head of these ribs. In fact, some sort of upper delimiter would be needed anyway, precisely because the ribs may contain lower delimiters.

The step from element 1.g to element 1.h might have been taken at any point after the lower delimiter was moved above the $if spine in element 1.f. When the immediate subexpression of an upper delimiter is a lower delimiter, there is no longer any quasishared graph between the two. So now both delimiters can be removed, with pointers to the i^{th} entry point of the upper delimiter redirected to point to the i^{th} child of the lower delimiter.

The other steps from element 1.f to element 1.i are simple reduction steps. Subsequent reduction down the right side of the graph will involve neither narrowing nor delimiter propagation since we insisted that the body of M contain no references to the formal parameter of $h. However, without quasisharing we would have to copy M, recomputing its reduction and reconsuming the space and time required to do so. By quasisharing M, we wait to see what parts of M *must* be duplicated for reduction to proceed, and only duplicate as absolutely necessary. In this case M will not refer to the lower delimiter, so the entire (large) subexpression may remain fully shared.

We specify all of the ways in which delimiters are to propagated through the graph in Fig. 2; elements (a) through (f) illustrate the upward propagation of lower delimiters.

1. For an application spine we break down the manipulation one step at a time. When a lower delimiter is the right child of an application node whose left child is a supercombinator (2.a), we can move the delimiter up without duplicating the supercombinator or placing an upper delimiter over it, since a supercombinator reference contains no children to refer to other lower delimiters[3]. Otherwise, the propagation of a lower delimiter up one side of an application node requires that a similarly tagged upper delimiter be placed atop the subexpression on the other side (2.b, 2.c). If similarly tagged delimiters are found on both sides of an application node, then both may be propagated at the same time (2.d).

2. Pair and binary union nodes may be treated similarly to application nodes in the event that one wishes to propagate lower delimiters above them (the three rules for each of these constructors are trivially similar to rules 2.b through 2.d, so in the interest of brevity we omit the former six from the figure). For singleton sets the manipulation is considerably simpler (2.e).

3. Propagating lower delimiters over quantifier abstractions is more complicated (2.f). The difficulty arises from duplicating the abstractions themselves: the subexpressions with which each new abstraction is paired will contain references to the quantifier over the

[3]In fact, this rule is subsumed by rule 2.l. We list it separately because upper delimiters above supercombinator references in the situation outlined here will *always* be removed, while those above other types of childless nodes, or above supercombinator references in a different context, might never be demanded, and thus would not have to be removed.

original abstraction. A correct duplication of an abstraction must make a new copy of the enclosed expression replacing references to the old quantifier with quantifier of the new abstractions. This mechanism for substitution is the same one required for distributing abstractions over binary unions. The need to make a substitution in this circumstance seems to be the main penalty of quasisharing, but we believe that the work saved in other circumstances will generally outweigh this cost.

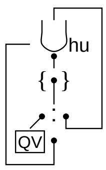
The cancelling of corresponding upper and lower delimiters is described in an additional single rule (2.g); when the upper and lower delimiters do *not* correspond to the same split quantifier, each propagates past the other, shown by (2.h).

Corresponding to moving lower delimiters up, any of several situations may require that we move upper delimiters down. For example, rule 2.b can place an upper delimiter in the middle of a spine: if a subsequent use of that spine has a supercombinator *below* the delimiter needing to access arguments *above* the delimiter, then the delimiter will have to be moved downward to facilitate the application of the supercombinator. We would also have to move an upper delimiter downward if it were the topmost graph node of an argument to $left or $right. These two situations are specific cases of the following general rule: *A non-quasishared access to the children of a quasishared node requires the unsharing of that node.* The graph manipulations for the downward propagation of upper delimiters conclude the series of rules in Fig. 2. Element 2.i gives the manipulation for application nodes, and again we omit the rendering of the rule for unions and pairs since except for the union or pair node they are identical to that for applications. Element 2.j shows the manipulation for singleton nodes; element 2.k, for abstractions, which again require substitutions; and element 2.l, for the trivial removal of upper delimiters from above nodes with no children.

Self-Reference. It is possible for a well-defined set to be based on a self-recursive set definition; for example, the two programs

$$\begin{array}{l}\texttt{\$S x} = \bigcup_{\texttt{y}\in\texttt{hu}}\{\texttt{<x,y>}\}\\ \texttt{\$main} = \bigcup_{\texttt{x}\in\texttt{hu}}\texttt{\$S x}\end{array}\qquad\text{and}\qquad\begin{array}{l}\texttt{\$S} = \bigcup_{\texttt{x}\in\texttt{hu}}\{\texttt{<x,\$S>}\}\\ \texttt{\$X x = \$dist (\$right x) (\$Y x)}\\ \texttt{\$Y x y} = \{\texttt{<\$left x, \$left y>}\}\\ \texttt{\$main = \$dist \$S \$X}\end{array}$$

denote the same set (viz., pairs of quantifiable values). Self-reference in a set described by an abstraction will lead to problems if it is implemented by a looped graph. Consider in the latter program the graph representation of `$S` by a graph with a simple loop:



. (2)

For an element x of this set, one would be forced to surmise from the graph that the expressions (`$left` x) and (`$left` y), $y \in$(`$right` x) denote the same quantifier, which is clearly not intended. An illustrative renaming of bound quantifier names gives

$$\begin{array}{l}\texttt{\$S} = \bigcup_{\texttt{x}\in\texttt{hu}}\{\texttt{<x, \$S>}\} = \bigcup_{\texttt{y}\in\texttt{hu}}\{\texttt{<y, \$S>}\}\\ \quad = \bigcup_{\texttt{x}\in\texttt{hu}}\{\texttt{<x, }\bigcup_{\texttt{y}\in\texttt{hu}}\{\texttt{<y, \$S>}\}\texttt{>}\}\ .\end{array}$$

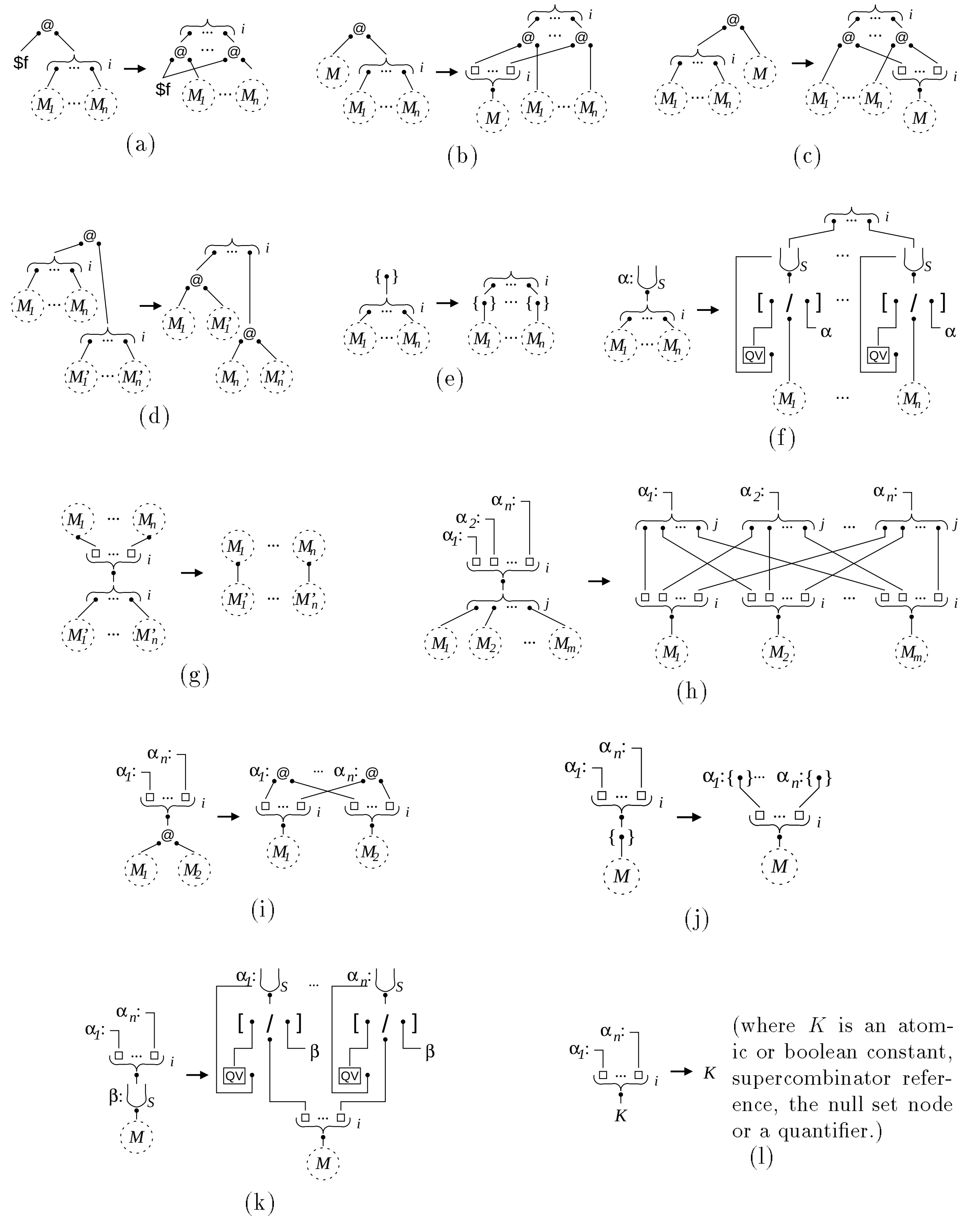
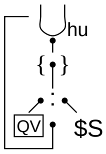


Figure 2: Samples of propagation of quasisharing delimiters: (a-f), upward propagation of lower delimiters; (g), cancellation of a matching upper/lower pair; (h), propagation of a non-matching upper/lower pair; (i-k), downward propagation of upper delimiters; (l), removal of upper delimiters.

So clearly it is incorrect to identify the two quantifiers; they range over the same set, but they do so independantly.

There is a similar problem for representing functions by unique graphic lambda expressions: one might be tempted to alter the body of the abstraction upon applying the function, but other references to the original, unapplied lambda expression would thus not be preserved. Therefore, a new body of the abstraction must be created for every distinct application [9, §12.1.3]. The situation for set abstractions is analogeous: nested occurances of a set should be separately build, so that each distinct quantifier refers to a distinct abstraction. So then the graph (2) above would be correctly built as



,

and the new call to `$S` expanded only upon need. Our previous restriction that a supercombinator may only contain a single set abstraction at the very leftmost point of its body facilitates this mechanism; if abstractions could occur at any point within the body, then this sort of expansion would be difficult to perform correctly, and in any event, would almost certainly waste space.

Though recursion is fundamental to function definition, it is much less likely that the user will want to define a self-containing set. Still, our language is untyped, and the domain-theoretical model by which it is defined [11] will give meaning a set isomorphic to one of its elements. Although such sets may seem bizarre, it is important that our abstract machine handle them correctly; because such sets are so exotic, efficiency in dealing with them is not of the highest priority. Therefore, we consider it reasonable to re-instantiate a recursively-applied supercombinator when necessary to ensure that each distinct usage of a parameterized set has its own distinct parameter. Of course, more space-efficient cyclic graphs can still be used to implement self-containing sets which do not introduce quantifiers.

Related Work

Graph reduction was first suggested as a language implementation technique by Wadsworth [12]. The first use of graph reduction for functional-logic programming in particular seems to have been by Reddy [10]. In his system the various occurrences of a subexpression arising from multiple uses of an abstraction parameter in a function body within a single nondeterministic choice branch are shared, but his system does not provide for sharing across alternative narrowings.

To our knowedge, ours is the first scheme which completely implements laziness in the usual functional programming sense. Loogan, Fraguas and Rodrígues-Artalejo [7] describe a technique for capturing one aspect of functional-style laziness — that only needed expressions should ever be evaluated — but since their technique does not allow sharing across alternative narrowings, the other aspect of laziness — that each needed expression must only be evaluated once — is lost. Hanus's survey [4] discusses contemporary implementations of functional logic languages, but again these techniques are not completely lazy, and do not provide sharing across narrowings. Moreover, some of these schemes restrict the use of higher-order function values, or else do not cleanly define such values in their declarative semantics; the set-based scheme does allow general use of function values and partial applications in the standard applicative programming sense (which does not, however, include higher-order unification).

Liu [6] presents a lazy functional logic graph reduction scheme which does allow some sharing across or-parallel branches. His technique is drawn mainly from techniques for AND/OR tree rewriting. Under his scheme, references across or-parallel branches to some unevaluated expression may share a single copy of the graph of that expression. Moreover, simple reductions within the expression which do not bind shared logic variables may update the shared graph. As here, the motivation is that even where copying is inevitable, reductions which would be common to all the copies are best done before copying. But Liu's results are limited in that when a shared logic variable within such a shared graph must be bound, the tree of the *entire* shared expression must be duplicated; our technique restricts such duplication to subterms of shared expressions which are directly affected by refinements to quantifiers.

Conclusion

The quasisharing technique presented here is interesting in that it allows more sharing across distinct expressions created in narrowing than previous approaches, admits both call-by-name and call-by-value semantics as a natural part of its design, and moreover does both in a framework which is readily supportive of a compilation to fixed low-level code. Further improvements to an implementation of quasisharing graph reduction could be realized both from functional optimizations to reduce unnecessary graph construction and from logic analyses to reduce the need for narrowing. Of course, development of such formal analysis and transformation techniques would require a more rigorous development of the graph calculus we present here: a confluence proof for the general calculus; a full specification of the qualified outermost reduction order and a standardization proof for it; soundness and adequacy results for Silbermann and Jayaraman's model [11], or some other model useful for particular analyses.

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 such compilation, and have designed an extension to the G-machine [8]—dubbed the NG-machine, for <u>N</u>arrowing <u>G</u>-machine—which uses quasisharing, and which we are currently implementing.

Acknowledgements. The authors would like to thank Boum Belkhouche, Robert Goldman, Martin Odersky and several anonymous reviewers for their comments on this paper and its earlier states.

References

[1] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1981.

[2] J. Darlington, A. J. Field, and H. 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.

[3] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, 1989.

[4] Michael Hanus. The integration of functions into logic programming: From theory to practice. To appear in the *Journal of Logic Programming*. Also available as Technical Report MPI-I-94-201, Max-Planck-Institut für Informatik, Saarbrücken.

[5] 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.

[6] 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.

[7] Rita Loogen, F. L. Fraguas, and Mario Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Lecture Notes in Computer Science 714*, pages 184–200. Fifth International Symposium on Programming Language Implementation and Logic Programming, Springer-Verlag, August 1993.

[8] 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.

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

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

[11] 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.

[12] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.