An Abstract Machine for Compiling Set-Based Functional Logic Languages

Revised Version

John Maraist

September 10, 1994

Written in partial fulfillment of the thesis requirement for the degree of Master of Engineering from the School of Engineering, Tulane University, New Orleans, LA, and submitted on July 15, 1993. Original thesis committee: Drs. F. S. K. Silbermann, B. Belkhouche and R. P. Goldman.

Foreword to the Revised Version

This version of my thesis corrects grammatic, typographical and spelling errors contained in the original, and resolves a number of problems in the original design of the NG-machine. A complete list of major changes may be found in the separate addenda sheet. Thanks in particular to John Kelly for pointing out several typographical errors.

Acknowledgements

I would like to express my gratitude to the members of my committee, Dr. Boumediene Belkhouche, Dr. Robert Goldman and Dr. Frank Silbermann, for their time and effort in reviewing this thesis. I am particularly indebted to Dr. Silbermann, my supervisor, for exposing me to functional-logic programming, for helping me to grapple with basic ideas, and later, for trusting in my instincts about this project and for helping me to sort out the subtler details in the work. Dr. Belkhouche pointed out several errors and ill-phrased statements, especially in the sections reviewing the semantics of PowerFuL. Dr. Goldman made extensive and insightful comments about my writing style and presentation and suggested background areas which needed further or better explanation; his comments greatly improved the overall readibility of the thesis.

Over the five years I have been at Tulane, Dr. Johnette Hassell and Dr. Mark Benard have generously and patiently given me both their personal and professional support. Without their assistance I would have succumbed long ago to red tape, system faults and poor scheduling, and would have had a far less enriching experience.

Although none were directly involved in this thesis, I owe very much to three teachers whom I encountered along the way: Prof. John Hughes, who introduced me to the curious area of computer science called functional programming; Mr. Frank Ducrest, who convinced me that yes, there is something to computers beyond BASIC; and Mme. Cecile Garboushian, who taught me to speak French and to have *passion* for any work I would want to be proud of.

Bill Thomason, the department's system administrator, helped me to solve innumerable problems with language and system software.

For making the late nights and long hours tolerable, for wonderful lunches at the sushi bar down at Riverbend, for some of the best reasons to take an afternoon off, or for having something to do after work, no one could be better than the CS grad students and the 'TUL staff.

Very little of my education would have been completed without my parents, who provided me with a solid background and their personal and material support.

Finally, and most fondly, I am grateful to Paula Ouder and James Hartman for their support, their friendship and their encouragement at the times when I needed it most.

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

Contents

List of Tables

List of Figures

Editorial Notes

A number of languages and notations are used in this thesis, and are distinguished in part by font. The syntax of PowerFuL and the syntax of the abstract machine code will both be represented in `teletype`. The semantics of PowerFuL and the intermediate language resembling it will be represented in **boldface**; the names of sets and domains of values over which the semantics range will be represented in upper-case boldface, except when the names of these sets are referenced as part of a construct normally printed in other layout. Node tags in the abstract machine, and other literal abstract machine constructs, are written in sans-serif lettering. The names of supercombinators, reflecting previous printing conventions, will be preceeded with a dollar sign ($). Metavalues will be printed in *italics*. Names of functions translating from language to language are written in $\mathcal{SCRIPT}$.

L'enfant joua. Il reprit la sonatine au meme rythme que précédemment et, la fin de la leçon approchant, il la nuança comme on le désirait, moderato cantabile.

— Quand il obéit de cette façon, ça me dégoûte un peu, dit Anne Desbaresdes. Je ne sais pas ce que je veux, voyez-vous. Quel martyre.

L'enfant continua néanmois à bien faire.

— Quelle éducation lui donnez-vous là, Madame Desbaresdes, remarqua la dame presque joyeusement.

Marguerite Duras, *Moderato Cantabile*

Passent les jours et passent les semaines
Ni temps passé
Ni les amours reviennent
Sous le pont Mirabeau coule la Seine

Guillaume Apollinaire, "Le Pont Mirabeau"

Dogs say cats love too much, are irresponsible,
are changeable, marry too many wives,
desert their children, chill all dinner tables
with tales of their nine lives.
Well, they are lucky. Let them be
nine-lived and contradictory,
curious enough to change, prepared to pay
the cat price, which is to die
and die again and again,
each time with no less pain.
A cat minority of one
is all that can be counted on
to tell the truth. And what cats have to tell
on each return from hell
is this: that dying is what the living do,
that dying is what the loving do,
and that dead dogs are those who do not know
that dying is what, to live, each has to do.

Alastair Reid, "Curiosity"

Chapter 1

Introduction

This thesis addresses the implementation of functional languages extended to include sets for logic programming. The expressive abilities of functional and logic programming are both well-known: functional programming allows values to be specified for computation with no need for a complete enumeration of every step of calculation; logic programming allows references to values according to properties and conditions to be met by those values. Combinations of the two styles allow an even more expressive notation not completely natural to either individual style, such as specifying a certain manipulation of all values satisfying a particular constraint. Several different functional-logic languages have been proposed, and several different implementation schemes have been outlined. By and large, both the languages and the implementation methods tend to draw more heavily from traditional logic programming languages than from traditional functional languages, favoring logic variables and clause unification as language constructs, and compilation based on the Warren Abstract Machine (WAM), SLR clause resolution, and some term rewriting. However, these techniques bring with them some severe restrictions on the functional programming style. Lazy evaluation is not native to traditional logic programming, and machines derived from these techniques such as BABEL [5, 3, 1] implement only a weakened form of laziness [1], if any at all. Similarly, logic programming traditionally allows higher-order values only in quite restricted contexts. Higher-order functions are not part of the basic design of BABEL [5], and while they are hacked into some later implementations [3] the arguments for correctness and other properties presented for the first machine do not include these extensions. Moreover, as Mück points out [6], implementations based on the WAM bear the weight of the WAM's complexity, which leads to large, intricate designs that are difficult to debug, improve and reason about formally. Mück extended the Categorical Abstract Machine (CAM), which compiles functional languages, to perform unification of terms, and thus defined the CAM Extension for Functional-Logic Programming (CAMEL). However, both the CAM and the CAMEL use fully strict evaluation semantics. While Mück does point out that CAMEL could include laziness via the `promise` and `force` constructs of LISP, such a mechanism will greatly obscure the syntax of programs and will also complicate the abstract machine. While not directly addressing functional-logic languages, Hughes makes a very good argument for the inclusion of laziness and higher-order constructs as fundamental aspects of a language's design [2].

Silbermann and Jayaraman's language PowerFuL [10] approaches the paradigm differently. Rather than explicitly including logic variables is the language, PowerFuL phrases logic programming in terms of set abstraction. For example, a Prolog programmer could find values which make some clause `goal` true by asking the system

```
?- goal(X).
```

A PowerFuL programmer would find values that make a boolean-valued function `goal` return a value of `true` by computing the program

```
{ x: x∈hu, (goal x) }
```

The identifier `hu` denotes the Herbrand universe of first-order terms, from which Prolog implicitly enumerates values. Unlike its Prolog counterpart, there are no restrictions on the use of `x` within the scope of its definition. Many Prolog predicates cannot accept logic variables as particular arguments to certain predicates. Short of type restrictions, PowerFuL will not distinguish between identifiers bound by set abstraction from identifiers used as formal parameters or local definitions.

In contrast with previous work, the approach introduced here draws implementation techniques from those associated with functional, rather than logic, programming. Since the fundamental design of PowerFuL is purely functional, it is appropriate to begin with an implementation method from those used for functional languages. The implementation described here uses the technique of graph reduction to design an abstract machine whose code is amenable to translation into a machine-executable form, such as assembly language or C code. A specialized form of graph narrowing is used to extend ordinary graph reduction for the required computations.

This thesis is organized as follows: Chapter 2 discusses the relevant background material, including a review of PowerFuL (§2.1), the compilation of pure functional languages (§2.2), one specific model of compilation (§2.3) which will be extended for functional-logic languages in a later chapter and the mapping of a purely functional subset of PowerFuL onto the abstract machine of that model (§2.4). Chapter 3 presents an extension to graph reduction to allow narrowing of the sets upon which set abstractions are based (§3.1-§3.4). Chapter 4 first justifies the choice of a particular functional language abstract machine as a starting point for the new implementation. This abstract machine is then evolved to use the extended graph model (§4.1-§4.5), and the translation schemes of §2.4 are adapted for the set-based syntactic constructs of PowerFuL and the mechanism of the new abstract machine (§4.8). Finally, Chapter 5 concludes the presentation and suggests further work.

Chapter 2

Background

This chapter presents previous research which forms the background to this thesis. We discuss the PowerFuL language in §2.1, some relevant issues in the compilation of lazy functional languages in §2.2, the basic design of a G-machine — slightly modified with an eye to the remainder of this thesis, but largely indistinguishable from the standard presentations — in §2.3, and the translation of a generic functional language to this abstract machine in §2.4.

2.1 PowerFuL

This section describes the language PowerFuL. The source of the material presented in this section is Silbermann and Jayaraman's paper presenting the language [10]. Figure 2.1 shows a BNF grammar for PowerFuL. The core of the language's syntax is similar to standard lazy functional languages:

- Atomic and boolean values are referred to by name.
- Functions are represented using the notation of the basic untyped lambda-calculus.
- Application of a function to an argument is written as the juxtaposition of the two.
- Primitive functions construct pairs (`cons`) and return elements of pairs (`car`, `cdr`); determine whether an item is an atom, boolean value, pair, function or set (`atom?`, `bool?`, `pair?`, `func?` and `set?`, respectively); select between alternative values (`if`); and determine a weak form of semantic equality between arbitrary expressions (`=`), and equality to the distinguished atom `'nil`.
- Local declarations are made with the `let` construct. An expression of the form

  `let` *id* `be` *val* `in` *expr*

  takes the value of *expr* where references to *id* in *expr* are given the value of *val*. Similarly, recursive definitions are made with the `letrec` construction.

Silbermann and Jayaraman add logic programming to this functional language with sets. The syntax of sets in PowerFuL is as follows:

- The empty set is represented by the distinguished operator ϕ.
- Singleton sets are constructed by enclosing expressions in curly braces: where *expr* is a well-formed formula denoting some value, $\{$*expr*$\}$ denotes the singleton set containing that value.
- The $\cup$ operator is used to denote the union of two sets.

```
     expr ::=  (expr)
             | TRUE | FALSE
             | A_1 | ...| A_n
             | λ identifier. expr
             | expr expr
             | cons(expr)
             | car(expr) | cdr(expr)
             | atom?(expr) | bool?(expr) | pair?(expr)
             | func?(expr) | set?(expr)
             | if(expr, expr, expr)
             | expr= expr | null?(expr)
             | let id be expr in expr | identifier
             | letrec id be expr, ..., id be expr in expr
             | ϕ | {expr} | ∪(expr, expr) | hu
             | {expr: conditions}
conditions ::=  condition, conditions | condition
 condition ::=  expr | id∈ expr
```

Figure 2.1: The syntax of PowerFuL.

- Set abstraction is used to derive sets from other sets. The expression $\{$***expr***: ***id***$\in$ ***set***$\}$ consists of the set of all values denoted by the expression *expr*, which possibly contains references to the identifier *id*, where *id* is taken to be any value of the set *set*. Many *id/set* abstractions may be specified, and are written in sequence, separated by commas. Restrictions on the values which identifiers may take are specified by boolean expressions included in the list of *id*s and *set*s; only those assignments of values to identifiers which cause the expressions to denote boolean truth are used in generation.

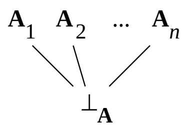
- The Herbrand universe of first-order terms is referred to by the distinguished identifier `hu`. The Herbrand universe contains atoms, boolean values and pairs of elements of the Herbrand Universe.

PowerFuL programs denote values ranging over the domain **D** which is the least solution to the equation

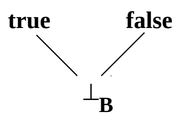
$$\mathbf{D} = (\mathrm{A} + \mathrm{B} + (\mathrm{D} \times \mathrm{D})_\perp + [\mathrm{D} \rightarrow \mathrm{D}] + \mathcal{P}(\mathrm{D}))_\perp$$

where:

- **A** is the domain of atomic elements, plus a bottom element $\perp_A$, under the flat ordering:



- **B** is the domain of boolean values, plus a bottom element $\perp_B$, under the flat ordering:



- $(\mathbf{X} \times \mathbf{Y})$ is the Cartesian product domain of **X** and **Y**, with the usual ordering of

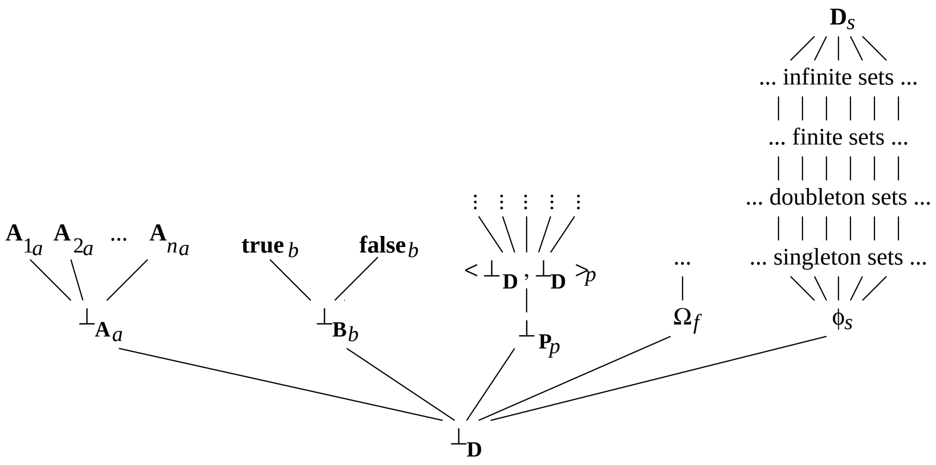
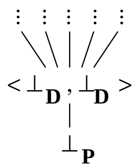


Figure 2.2: The powerdomain **D**.

$$\langle x_1, x_2 \rangle \sqsubseteq \langle y_1, y_2 \rangle \iff (x_1 \sqsubseteq y_1) \wedge (x_2 \sqsubseteq y_2)$$

$(\mathbf{X} \times \mathbf{Y})_\perp$ adds a new bottom element $\perp_P$ to $(\mathbf{X} \times \mathbf{Y})$:



- $[\mathrm{X} \rightarrow \mathrm{Y}]$ is the domain of continuous functions from **X** to **Y** with the usual ordering

  $$f \sqsubseteq g \iff (\forall x)\, f(x) \sqsubseteq g(x)$$

  which identifies a least element $\Omega = \perp_{\mathbf{F}} = (\lambda x.\ \perp_{\mathbf{D}})$.

- $\mathcal{P}(\mathrm{D})$ denotes the powerdomain of **D**, under the lower ordering, by which the empty set is the least element.

- The sum operation $(X_1 + \ldots + X_n)_\perp$ denotes the separated sum, where the bottom elements of the domains to be summed are kept distinct, with a new bottom element as the least element of the sum. Figure 2.2 depicts this sum.

D itself is a lower powerdomain; the choice of ordering is justified by Silbermann and Jayaraman [10]. The construction of powerdomains is surveyed by Main [4].

The relations $\mathcal{D}$ and $\mathcal{E}$ map PowerFuL programs into **D**; we list these rules in full in Figure 2.3. The mapping of the purely functional core of the language to **D** is straightforward:

- The embedding of the subdomains into **D** is expressed by the existance of *embedding markers*[1], written as subscripts. The projection of embedded subdomain elements back into their respective domains is expressed by a *retract*, written as a vertical bar followed by the initial of the subdomain into which the projection is performed.

[1]Usually, embedding markers are referred to as *tags* [10]. However, "tag" is also used to describe one component of the abstract machine. To avoid confusion, "embedding markers" is used in this thesis for the former meaning.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathbf{A}_i]\!]\,\rho & = & \mathbf{A}_{i\,a} \\
\mathcal{E}[\![\texttt{true}]\!]\,\rho & = & \mathbf{TRUE}_b \\
\mathcal{E}[\![\texttt{false}]\!]\,\rho & = & \mathbf{FALSE}_b \\
\mathcal{E}[\![(expr)]\!]\,\rho & = & \mathcal{E}[\![expr]\!]\,\rho \\
\mathcal{E}[\![\texttt{cons}(expr1, expr2)]\!]\,\rho & = & (\langle \mathcal{E}[\![expr1]\!]\,\rho, \mathcal{E}[\![expr2]\!]\,\rho\rangle)_p \\
\mathcal{E}[\![car(expr)]\!]\,\rho & = & \mathbf{left}\ (\mathcal{E}[\![expr_1]\!]\,\rho|p) \\
\mathcal{E}[\![cdr(expr)]\!]\,\rho & = & \mathbf{right}\ (\mathcal{E}[\![expr_1]\!]\,\rho|p) \\
\mathcal{E}[\![if(expr1, expr2, expr3)]\!]\,\rho & = & \mathbf{if}(\mathcal{E}[\![expr1]\!]\,\rho|b, \mathcal{E}[\![expr2]\!]\,\rho, \mathcal{E}[\![expr3]\!]\,\rho) \\
\mathcal{E}[\![bool?(expr1)]\!]\,\rho & = & (\mathcal{E}[\![expr1]\!]\,\rho \in b)_b \\
\mathcal{E}[\![atom?(expr1)]\!]\,\rho & = & (\mathcal{E}[\![expr1]\!]\,\rho \in a)_b \\
\mathcal{E}[\![pair?(expr1)]\!]\,\rho & = & (\mathcal{E}[\![expr1]\!]\,\rho \in p)_b \\
\mathcal{E}[\![func?(expr1)]\!]\,\rho & = & (\mathcal{E}[\![expr1]\!]\,\rho \in f)_b \\
\mathcal{E}[\![set?(expr1)]\!]\,\rho & = & (\mathcal{E}[\![expr1]\!]\,\rho \in s)_b \\
\mathcal{E}[\![expr1 = expr2]\!]\,\rho & = & (\mathbf{DeqD?}(\mathcal{E}[\![expr1]\!]\,\rho, \mathcal{E}[\![expr2]\!]\,\rho))_b \\
\mathcal{E}[\![null?(expr)]\!]\,\rho & = & (\mathbf{DeqA?}(\mathcal{E}[\![expr]\!]\,\rho, \texttt{nil}))_b \\
\mathcal{E}[\![identifier]\!]\,\rho & = & \rho(identifier) \\
\mathcal{E}[\![\texttt{let}\ id\ \texttt{be}\ expr_1\ \texttt{in}\ expr_2]\!]\,\rho & = & \mathcal{E}[\![expr_2]\!]\,\rho[(\mathcal{E}[\![expr_1]\!]\,\rho)/id] \\
\mathcal{E}[\![\texttt{letrec}\ defs\ \texttt{in}\ expression]\!]\,\rho & = & \mathcal{E}[\![expression]\!]\,(\mathcal{D}[\![defs]\!]\,\rho) \\
\mathcal{E}[\![\lambda\ id\ .\ expr]\!]\,\rho & = & (\lambda\ x.\ \mathcal{E}[\![expr]\!]\,\rho[x/id])_f \\
\mathcal{E}[\![expr_1\ expr_2]\!]\,\rho & = & (\mathcal{E}[\![expr_1]\!]\,\rho)|f\ \mathcal{E}[\![expr_2]\!]\,\rho \\
\mathcal{E}[\![\phi]\!]\,\rho & = & \boldsymbol{\phi} \\
\mathcal{E}[\![\texttt{hu}]\!]\,\rho & = & \mathbf{hu} \\
\mathcal{E}[\![\texttt{U}(expr_1, expr2)]\!]\,\rho & = & ((\mathcal{E}[\![expr_1]\!]\,\rho)|s\ \cup\ (\mathcal{E}[\![expr2]\!]\,\rho)|s)_s \\
\mathcal{E}[\![\{\ expr\!:\}\ ]\!]\,\rho & = & (\{\mathcal{E}[\![expr]\!]\,\rho\})_s \\
\mathcal{E}[\![\{expr : condition, qlist\}]\!]\,\rho & = & (\mathbf{IF}_{\mathrm{S}}(\mathcal{E}[\![condition]\!]\,\rho|b, \mathcal{E}[\![\{expr : qlist\}|s]\!]\,\rho, \phi))_s \\
\mathcal{E}[\![\{expr : id \in genrtr, qlist\}]\!]\,\rho & = & \left(\lambda X.((\mathcal{E}[\![\{expr : qlist\}]\!]\ \rho\left[\dfrac{X}{id}\right])|s)\ ^{+}\ (\mathcal{E}[\![genrtr]\!]\,\rho)|s\right)_s \\
\mathcal{D}[\![id\ \texttt{be}\ expr]\!]\,\rho & = & \rho[(\mathbf{fix}(\lambda\ X.\ \mathcal{E}[\![expr]\!]\,\rho\left[\dfrac{X}{id}\right]))/id] \\
\mathcal{D}[\![id\ \texttt{be}\ expr, defs]\!]\,\rho & = & \mathcal{D}[\![defs]\!]\,(\rho[\mathbf{fix}(\lambda\ X.\ \mathcal{E}[\![expr]\!]\,(\mathcal{D}[\![defs]\!]\,\rho[X/id]))/id])
\end{array}
$$

Figure 2.3: The semantics of PowerFuL.

- Recursion is expressed in terms of the lowest fixed point operator **fix**.
- The Herbrand Unvierse, **hu** is defined to be the least domain satisfying the equation

$$\mathbf{hu} = (\mathbf{A}_a + \mathbf{B}_b + ((\mathbf{hu} \times \mathbf{hu})_\perp)_\perp)_p.$$

- Silbermann and Jayaraman give full definitions of the domain primitives used here [10].

The meaning of the set-based operators relies on the $^+$ primitive, defined

$$\begin{aligned} f^+\phi &= \phi \\ f^+\{x\} &= f\ x \\ f^+(X \cup Y) &= (f\ X) \cup (f\ Y) \end{aligned}$$

Silbermann and Jayaraman give an operational semantics for PowerFuL as a sort of optimized denotational semantics [10]. Rather than generate elements of the Herbrand Universe one element at a time, we can, without loss of expressiveness, restrict occurances of the identifer `hu` to syntactic contexts of the form

$$\{\mathit{expr}\colon\ \ldots,\ \mathit{id} \in \mathtt{hu},\ \ldots\}$$

We then modify the translation rule for this sort of expression to translate the syntax to an equivalently-valued expression, but using a different notation[2]:

$$\mathcal{E}[\![\{expr:\ id \in \mathbf{hu}, qlist\}]\!]\,\rho = \bigcup_{X \in \mathbf{hu}} \mathcal{E}[\![\{expr:\ qlist\}]\!]\,\rho[X/id]$$

The following identities are true with respect to this notation:

1. $\bigcup_{x\in S} \phi \ = \ \phi$
2. $\bigcup_{x\in S}(X \cup Y) \ = \ \bigcup_{x_1 \in S} X[x_1/x] \cup \bigcup_{x_2\in S} Y[x_2/x]$
3. $f^+ \bigcup_{x\in S} E \ = \ \bigcup_{x\in S} (f^+E)$

   These three axioms describe the way unions generated over sets interact with the other set primitives.
4. $\bigcup_{x\in\mathbf{hu}} E \ = \ \left(\bigcup_{x_1\in\mathbf{atoms}} E[(x_1)_a/x]\right) \cup \left(\bigcup_{x_2\in\mathbf{bools}} E[(x_2)_b/x]\right) \cup \left(\bigcup_{x_3\in\mathbf{hu}} \bigcup_{x_4\in\mathbf{hu}} E[\langle x_3, x_4\rangle_p\,/x]\right)$
5. $\bigcup_{x\in\mathbf{hu}} \ = \ \bigcup_{y\in\mathbf{hu}\setminus\{m\}} E[y/x] \cup E[m/x]$, fixed $m \in \mathbf{hu}$.
6. $\bigcup_{x\in\mathbf{atoms}} \ = \ \bigcup_{y\in\mathbf{atoms}\setminus\{a\}} E[y/x] \cup E[a/x]$, fixed $a \in \mathbf{atoms}$.
7. $\bigcup_{x\in\mathbf{bools}} E \ = \ E[\mathbf{true}/x] \cup E[\mathbf{false}/x]$

   These identities provide a semantic analog to graph narrowing, which we will use in conjunction with a parallel-outermost computation rule to compute the value of a PowerFuL program's semantic meaning. The computation rule is sufficient for the functional aspects of the language, and the axioms apply when non-grounded quantifiers inhibit reduction.

For example, consider the expression

$$\bigcup_{x\in\mathbf{hu}} \mathbf{if}(x \in \mathbf{atoms},\ \{x\},\ \phi)$$

[2] This is a variation on Silbermann and Jayaraman's notation [10]. What is written here as $\bigcup_{x\in\mathbf{hu}} expr$ would be written there as $hu(X).expr$.

As it is, the expression is not in normal form, but the expression $x \in \mathbf{atoms}$ is irreducible because x has no domain tag. But axiom 5 shows that that expression is equivalent to the following expression:

$$\bigcup_{x \in \mathbf{atoms}} \mathbf{if(}x_a \in \mathbf{atoms},\ \{x_a\},\ \phi\mathbf{)} \ \cup\ \bigcup_{y \in \mathbf{bools}} \mathbf{if(}y_b \in \mathbf{atoms},\ \{y_b\},\ \phi\mathbf{)}$$

$$\cup\ \bigcup_{z \in \mathbf{hu}} \bigcup_{w \in \mathbf{hu}} \mathbf{if(}\langle z, w\rangle_p \in \mathbf{atoms},\ \{\langle z, w\rangle_p\},\ \phi\mathbf{)}$$

The second and third components of this union reduce to ϕ: respectively,

$$\bigcup_{y \in \mathbf{bools}} \mathbf{if(}y_b \in \mathbf{atoms},\ \{y_b\},\ \phi\mathbf{)} \ =\ \bigcup_{y \in \mathbf{bools}} \phi \ =\ \phi$$

and

$$\bigcup_{z \in \mathbf{hu}} \bigcup_{w \in \mathbf{hu}} \mathbf{if(}\langle z, w\rangle_p \in \mathbf{atoms},\ \{\langle z, w\rangle_p\},\ \phi\mathbf{)} \ =\ \bigcup_{z \in \mathbf{hu}} \bigcup_{w \in \mathbf{hu}} \phi \ =\ \bigcup_{z \in \mathbf{hu}} \phi \ =\ \phi$$

So the entire expression reduces to

$$\bigcup_{x \in \mathbf{atoms}} \{x_a\},$$

which is in normal form.

2.2 Compiling Lazy Functional Languages

This section discusses two issues in the implementation of functional programming languages especially relevant to this thesis. Early functional languages were usually interpreted, rather than compiled, which led to severe performance penalties. The lack of compilation arose in part from a lack of a good compilation technique. The notion of *graph reduction*, and hardware improvements to make the use of the concept feasible, provided inspiration for the development of new implementation techniques. Many implementations of graph reduction — including the model drawn from here — require that the occurance of free variables in the bodies of functions be restricted. The process of *lambda-lifting* allows a program to be transformed into an equivalent series of definitions called *supercombinators*, which meet the appropriate restrictions.

2.2.1 Graph Reduction

Graph reduction is a method of programming language implementation introduced for the λ-calculus by Wadsworth [11]. The basic idea is to represent a program in memory by a directed graph derived from its syntax. Computation of the program then becomes the task of reducing the expression denoted by the graph into an appropriate normal form. Graph reduction is an appropriate implementation technique for languages where the value to be computed is explicity denoted by the text of the program, rather than languages which specify steps to be executed to produce the value; in particular, graph reduction is useful in implementing pure, lazy functional languages.

In and of itself, graph reduction is not necessarily an efficient technique. A naïve graph reducer is no more than an interpreter operating on the graph of a program. An efficient implementation of graph reduction entails making the graph itself executable. For example, an efficient approach to a graph node representation of an application of a function to an argument would be to place executable code with (or as) the representation of the function, and to pass the argument to this code. Presumably, the code would not only be able to compute the result of the function for the argument, but would also know how to update the original application node with its result for the given argument.

There are several implementations of lazy functional languages based in some way on graph reduction, but most have certain aspects in common.

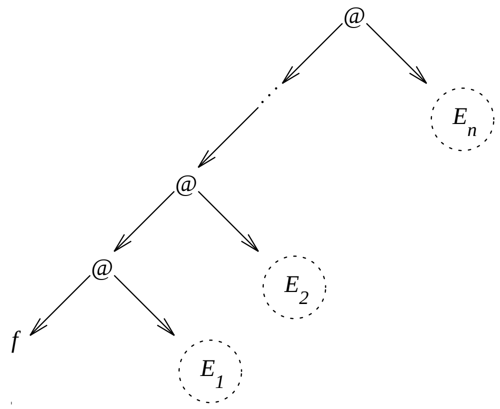


Figure 2.4: An application spine.

- Barring optimizations, data are given a *boxed* representation. That is, the primary method for representing expressions, including atomic information such as numbers, characters, boolean values and literal atoms is *not* (say, in the case of numberic values) the number itself, but is instead a pointer to a node containing that number, and *which is specifically tagged as containing numeric data.* This is necessary because (short of tagging pointers, which would require still more space) it is not possible to distiguish pointer values from literal numeric values, or numbers from character codes. Clearly, boxed values are not an efficient implementation for arithmetic; for this reason many optimizations are aimed at allowing unboxed values to be used, if not throughout the computation than at least during actual arithmetic operations.

- The operation of the graph reducer, whether interpretive or compilation-based, is a simple loop:

  Beginning from the top of the graph,

  1. Reduce the expression to some appropriate normal form.
  2. Print the information in the top node.
  3. Recur on this node's children.

  Of course, this loop may not terminate for structures which can be evaluated to arbitrary size.

- A less universal but still common property of graph reduction implementations is the occurance of a *spine* in the graph. Although the graph model for function application is usually application of a function to a *single* argument, many functions will take more than one argument. Thus, structures such as that in Figure 2.4 will frequently appear in the graph. Commonly, '@' is used to depict an application node. In the figure, the items labeled E_1, E_2, etc., denote subgraphs, and f denotes some representation of a function. Textually, the expression denoted by this bit of graph is written

  $$(\ldots((f\ E_1)\ E_2)\ \ldots)\ E_n)$$

  Since graph reduction proceeds from the outermost node, the reducer would traverse this spine, saving the "ribs" as future arguments, and looking for a function at the bottom of the spine. This process is called *unwinding the spine.* The particulars of manipulating the spine vary among implementations, but share a general outline. The evaluation mechanism reduces an expression by repeating the following steps:

1. Traverse the spine,
2. find a function of arity n (*i.e.*, a function requiring n arguments) at the base,
3. pare the lower n vertebrae from the spine, and taking their ribs as arguments,
4. execute the function on those arguments, and
5. update the spine with the result.

The repitition can cease when the expression no longer has an application node as its outermost graph element (which usually corresponds to the desired normal form).

Implementations of graph reduction which do not build spines are said to be *spineless*. Spineless machines are not considered in this thesis because the method for incorporating logic programming relies on the manipulation of spines.

- Implementations of graph reduction — both formal descriptions and actual compilers — will rely on some form of dynamic storage area, usually a heap, which will require garbage collection, and will also have at least one, and often two, stacks, whose use varies.

2.2.2 Free Variables & Supercombinators

Free variables within the bodies of local function definitions are an impediment to these functions' compilations. To produce a good compilation of functional languages, and in particular one which is less interpretive, individual functions in the source program must be translated into fixed sequences of (abstract) machine code. Free variables within the body of a function make this process difficult, because the compiler may not be able to determine exactly how and where these values will be represented and stored at the time the function is invoked. This problem does not apply for the formal parameters of the function, which are represented as the subgraphs along the spine of application nodes as described above. Nor does it apply to globally accessible constant values: these may be stored in reserved locations in the heap at the beginning of the execution of the program, and referenced in those locations. Note that a "constant value" may be a function, or at least a function with no free variables.

Consider the following nested function definitions:

```
let f = λx. let g = λy.(...expression involving x, and possibly also y...)
            in (...expression involving g and x...)
```

The value bound to `g` is a function, and (as for any function) it will be best to compile `g` to a fixed sequence of instructions. But taken by itself, the definition of `g` looks like this:

```
let g = λy.(...expression involving x, and possibly also y...)
```

It would be difficult, if not impossible, to determine how this `x` should be accessed. The value of `x` will be different among varying calls to `f`. If `f` and `g` are in fact mutually recursive, then when `g` calls `f` with a different argument, the recursive call to `f` will need to invoke a *different* version of `g`, one which reflects the new actual parameter.

This sort of construction is not a problem in traditional imperative languages. Compilers for those languages usually employ a stack-based model of compilation, where environment pointers may be used to resolve such references. Unfortunately, spine-based implementations cannot easily incorporate environments.

The solution to this problem is to eliminate free variables from function definitions. This process can be automated, and performed as an early phase of compilation. The process of removing free variables from function definitions is called *lambda-lifting*; several lambda-lifting algorithms have been described [7]. The result of lambda-lifting is a series of definitions denoting the same value as the original program, but where every definition is a *supercombinator*. A supercombinator is a simple function definition where every item mentioned in the definition is either

1. a literal constant value,
2. a formal parameter of the supercombinator, or
3. a supercombinator.

The body of a supercombinator definition may contain local bindings (with a `let` construct), recursive local definitions (a `letrec`) and applications, but no lambda-expressions.

The compilation schemes discussed in this thesis all assume that the source program is in the form of a sequence of supercombinator definitions. By convention, we preceed the names of supercombinators with a dollar sign ($).

We will make one further restriction to the structure of a supercombinator body: a set quantification $\bigcup_{\mathtt{x} \in S}[\,]$ may only appear as the leftmost term of a body. That is, every expression of the form

$$\bigcup_{\mathtt{x} \in S} E$$

shall be lifted to its own supercombinator. In addition, we will mark each supercombinator whose body is a quantifier with an exclamation point after the leading '`$`'. The reason for this additional restriction is discussed in §3.3.4

2.3 A Standard G-Machine

The basis for the abstract machine derived in this thesis is Peyton Jones's version of the G-machine [7], although the handling of global values and much of the notation is considerably different.

2.3.1 Basic G-Machine Constructs

A G-machine is a state machine defined by a finite sequence of *G-code instructions*, described below. The *state* of a G-machine is given by a quintuple,

$$(O, C, S, G, D)$$

where

- O is a finite sequence of output symbols,
- C is a finite sequence of G-code instructions,
- S is a stack of *addresses*,
- G is a graph modeled by a map from addresses to *graph nodes* and
- and D is the *dump*, a stack of pairs, each of which contains a finite sequence of code plus a stack of addresses.

Let K be the set of G-machine states. Except when explicitly describing an output-related process, we will write the state as a quadruple, omitting the output. It is often necessary to specify the first G-code instruction of an otherwise arbitrary sequence; a semicolon will separate particular instructions in a G-code sequence written from left-to-right (the semicolon will not be used for instructions written in a vertical sequence). Similarly, a colon will separate the first element of the stack or dump from the rest of the sequence. The notation

$$a \xrightarrow{G} N$$

means that the graph G maps the address a to the node N; the notation

$$G[a \mapsto N]$$

describes a graph which maps a to N, and all other addresses to wherever G maps them. The *initial state* of a G-machine given by the instruction sequence I is defined to be the tuple ([], I, [], ϕ, [])[3], where [] is an empty sequence. A G-machine state is said to be in a *final state* if it is of the form

$$(\texttt{Halt}; C, S, G, D)$$

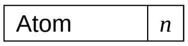
that is, one having reached the G-code instruction `Halt`.

The varieties of graph nodes reflect the different values representable by programs. Nodes are identified by a *node tag*. The different types of node, and their tags, are as follows:

- Atomic values are represented by one type of node, tagged as Atom and also containing a reference to the specific atom represented. In the text, atom nodes will be written

  Atom n

  and graphically represented as

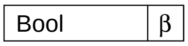
  

  where n is a number used to distinguish between atoms.

- Boolean values are similarly represented, with a tag to indicate the boolean value plus the truth value itself. Boolean-valued nodes will be written textually as

  Bool β

  and graphically as

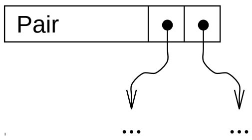
  

  where β is the boolean value represented.

- Pairs are represented by a single node with a unique tag, plus the addresses of the nodes for the two individual elements of the pair. Graph nodes representing pairs will be written in the text as

  Pair p_1 p_2

  where p_1 and p_2 are the addresses of the left and right elements of the pair, and graphically as

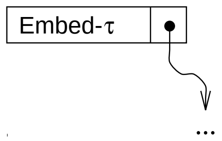
  where the two nodes point to the graphs of the pair's elements.

- The embedding of a value into **D** is represented by a node whose node tag represents the embedding marker, plus the address of the representation of the embedded subdomain value, displayed textually as

[3] Or if excluding the output component, (I, [], ϕ, []).

$$\text{Embed}-\tau\ e$$

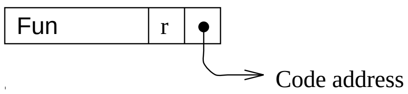
and graphically as



- Lambda abstractions in the original program will be compiled into G-code instruction sequences. These functions are referred to by another type of node, tagged as a function and including the arity of the function and the position in the G-code sequence defining the machine at which the compiled function begins. Function nodes will be expressed in the text as

$$\text{Fun}\ a\ pc$$

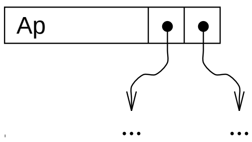
and graphically



- The application of one value (a function) to another (its argument) is modeled by a unique node, tagged as an application, with the addresses of the two subexpressions. The machine's application nodes will be written

$$\text{Ap}\ e_1\ e_2$$

in the text and

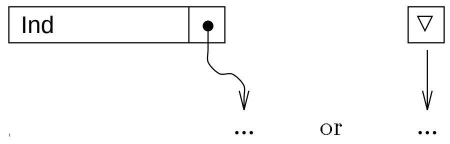


in graphics. In the literature of graph reduction, application nodes are often marked with the symbol "@"; in the interest of brevity we will also follow this convention for the graphic representation of application nodes.

- When in the course of reducing the graph we discover that the node at one address should contain the same tag and pointers as those which a second graph node already contains, it is in general preferable to simply make a reference to that node rather than to copy the particulars. Such a reference is made by an *indirection* node. In the text we write an indirection node to the node at address e as

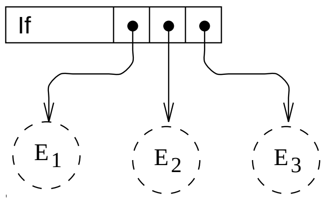
$$\nabla\ e$$

and graphically as either

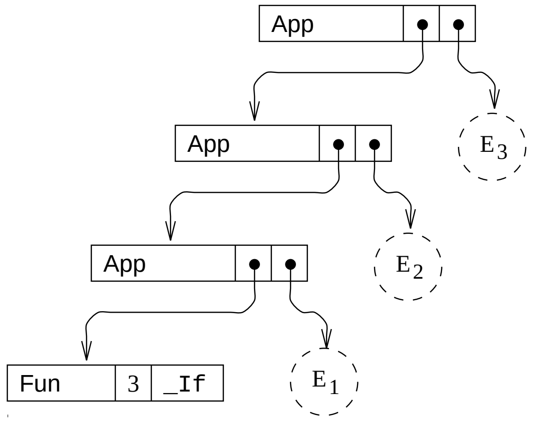


The G-machine instructions will "reference through" an indirection node and examine the node to which it points when necessary.

Many of the node types described in the graph models elsewhere in this thesis are not listed here: many nodes used in the more intuitive model descriptions are simply a more expressive notation, replaced in the abstract machine with combinations of the G-machine nodes. For example, we would represent the If node



in a G-machine by the following graph



The `_If` code sequence, described in full later, refers to a sequence of G-code which manipulates its three arguments as one would expect from the definition of the domain primitive **if**.

2.3.2 Basic G-Code Instructions and the Derivation Relation

The function $\vdash$, $\vdash$: $(K \to K)$, relates a G-machine state with the state derivable from it, if one exists. We define this function in terms of the G-code instruction at the beginning of the sequence in the first position of the state tuple, and in many cases also in terms of the other elements of the tuple. This section introduces the basic G-code instructions and begins to define the derivation relation.

- The `Eval` instruction forces the expression to which the address on the top of the stack points to be reduced to weak head-normal form. Once the expression is in weak head-normal form, the machine continues to the next instruction.
  - If the node which the address at the top of the stack indicates is already in weak head-normal form, as is the case for all node types *except* application nodes and functions of arity zero, we need to do no reduction:

$$(\texttt{Eval}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} N \vdash (C,\ a{:}S,\ G,\ D),$$

where

$$N \in \{\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \mathsf{Fun}\ n\ i,\ n > 0\ \}$$

- If the expression to be evaluated is an application node we must begin unwinding the spine of application nodes until we find a function node.

$$(\texttt{Eval}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ a_1\ a_2 \vdash (\texttt{Unwind},\ n{:}[\,],\ G,\ \langle S, C\rangle{:}\,D)$$

- If the expression is a zero-arity function, *viz.*, a constant, the machine jumps to the indicated sequence of G-code, saving its original configuration on the stack to be resumed later.

$$(\texttt{Eval}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Fun}\ 0\ C' \vdash (C',\ n{:}[\,],\ G,\ \langle S, C\rangle{:}\,D)$$

- Finally, indirection nodes must be dereferenced before they can be evaluated.

$$(\texttt{Eval}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Eval}; C,\ a'{:}S,\ G,\ D)$$

- The `Unwind` command attempts to unwind a spine of applications and apply the function at the base of the spine to the arguments along the ribs. Again, the item to which the top address on the stack refers will determine the specific operation of the instruction.
  - It is acceptable for a weak head-normal form expression aside from an unapplied function to be at the top of the stack for an `Unwind` command *only* if that reference to the expression is the only item on the stack. In such a case, there are simply no arguments to be applied and we expect to find some constant as the result, but otherwise the configuration would indicate the application of a non-function value, which is illegal. Thus we have the rule

$$(\texttt{Unwind}; C_0,\ n{:}[\,],\ G,\ \langle S, C\rangle{:}\,D) \vdash (C,\ n{:}S,\ G,\ D),$$

where

$$N \in \{\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \}.$$

  - Unwinding the spine for an application node involves placing the next element of the spine on the stack and unwinding again:

$$(\texttt{Unwind}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ n'\ m \vdash (\texttt{Unwind}; C,\ n'{:}n{:}S,\ G,\ D)$$

  - When unwinding reaches the bottom of the spine, there should be a function node there.
    - If the unwinding process has placed at least as many arguments on the stack as the arity of the function, then we can make the references to vertebrae into references to ribs, and execute the function:

$$\begin{gathered}(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_m{:}S,\ G,\ D),\ a_0 \xrightarrow{G} \mathsf{Fun}\ m\ C,\\ a_i \xrightarrow{G} \mathsf{Ap}\ a_{i-1}\ n_i,\ 1 \le i \le m\\ \vdash\ (C,\ n_1{:}\ldots{:}n_m{:}a_m,\ G,\ \langle S, \texttt{Unwind}\rangle{:}\,D)\end{gathered}$$

Zero-arity function do also fall into this category, with $m = 0$.

* On the other hand, a function applied to fewer arguments than its arity requires is already in weak head-normal form, and does not need to be reduced at all:

$$(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_k,\ G,\ \langle S, C\rangle{:}D),\ a_0 \xrightarrow{G} \mathsf{Fun}\ m\ C_1,\ k < m$$
$$\vdash\ (C,\ a_k{:}S,\ G,\ D)$$

* As is the case with the Eval command, indirection nodes must be dereferenced before they can be unwound:

$$(\texttt{Unwind}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Unwind}; C,\ a'{:}S,\ G,\ D)$$

- The Jump instruction simply executes a different sequence of code with the same stack, graph and dump configurations.

$$(\texttt{Jump}\ \ C; C_0,\ S,\ G,\ D) \vdash (C,\ S,\ G,\ D)$$

- The Push command places another copy of an address given somewhere in the stack on the top of the stack.

$$(\texttt{Push}\ \ n; C,\ a_0{:}\ldots{:}a_n{:}S,\ G,\ D) \vdash (C,\ a_n{:}a_0{:}\ldots{:}a_n{:}S,\ G,\ D)$$

- The Push Atom and Push Bool instructions place the indicated boxed basic value on the stack.

$$(\texttt{Push Atom}\ \ \alpha; C,\ S,\ G,\ D)\ \vdash\ (C,\ n{:}S,\ G[n \mapsto \mathsf{Atom}\ \alpha],\ D)$$
$$(\texttt{Push Bool}\ \ \beta; C,\ S,\ G,\ D)\ \vdash\ (C,\ n{:}S,\ G[n \mapsto \mathsf{Bool}\ \beta],\ D)$$

- The Push Global instruction places a reference to one of several distinguished global values onto the stack. We use this command to access global definitions established by the Set Global command (p. 17, below), corresponding to top-level definitions and built-in functions.

$$(\texttt{Push Global}\ \ i; C,\ S,\ G,\ D) \vdash (C,\ \gamma_i{:}S,\ G,\ D),$$

where γ_i represents the address of the location in the graph of the i^{th} global value.

- The Pop instruction is used to remove elements from the stack.

$$(\texttt{Pop}\ \ n; C,\ a_1{:}\ldots{:}a_n{:}S,\ G,\ D) \vdash (C,\ S,\ G,\ D)$$

- The Slide instruction is similar to the Pop instruction, except that it saves the top element of the stack, removes the number of elements indicated by its parameter, and then pushes the old top element onto the stack to be the new top element.

$$(\texttt{Slide}\ \ n; C,\ a_0{:}\ldots{:}a_n{:}S,\ G,\ D) \vdash (C,\ a_0{:}S,\ G,\ D)$$

- The Update instruction alters heap nodes to match that pointed to by a different pointer.

$$(\texttt{Update}\ \ n; C,\ a_0{:}\ldots{:}a_n{:}S,\ G,\ D) \vdash (C,\ a_1{:}\ldots{:}a_n{:}S,\ G[a_n \mapsto \nabla\ a_0],\ D)$$

- The Alloc instruction reserves heap and stack space to be filled in by subsequent code. We use this instruction to construct mutually recursive expressions.

$$(\texttt{Alloc}\ n; C,\ S,\ G,\ D) \vdash (C,\ a_1{:}\ldots{:}a_n{:}S,\ G[a_1 \mapsto \mathsf{Hole},\ \ldots,\ a_n \mapsto \mathsf{Hole}],\ D)$$

- The `MkAp` command takes n addresses from the stack and creates an application spine from them, leaving the single application node found at the top of the spine on top of the stack.

$$(\texttt{MkAp}\ n; C,\ a_0{:}\ldots{:}a_n{:}S,\ G,\ D) \vdash (C,\ e_n{:}S,\ G\left[\begin{array}{l} e_i \mapsto \mathsf{Ap}\ e_{i-1}\ a_i,\ i > 1 \\ e_1 \mapsto \mathsf{Ap}\ a_0\ a_1 \end{array}\right],\ D)$$

- The `MkPair` and `MkEmbed` τ instructions are similar to the `MkAp 1` instruction but create either a pair node or one of the tagged nodes. Of course, an application node would require further evaluation while these nodes necessarily mark an expression already in weak head normal form.

$$\begin{array}{rcl} (\texttt{MkPair}; C,\ a_1{:}a_2{:}S,\ G,\ D) & \vdash & (C,\ a_*{:}S,\ G[a_* \mapsto \mathsf{Pair}\ a_1\ a_2],\ D) \\ (\texttt{MkEmbed}\ \tau; C,\ a{:}S,\ G,\ D) & \vdash & (C,\ a_*{:}S,\ G[a_* \mapsto \mathsf{Embed}{-}\tau\ a],\ D) \end{array}$$

- There are two commands to set up globally referenced quantities: `Alloc Globals` directs the machine to set aside the specified number of heap slots for the storage of global values; `Set Global`, which takes three parameters, n, k and a, places in the n^{th} reserved heap space a node tagged as a function of arity k executed by the G-code sequence at location a in the sequence of code defining the particular machine.

$$\begin{array}{rcl} (\texttt{Alloc Globals}\ n; C,\ S,\ G,\ D) & \vdash & (C,\ S,\ G[\gamma_0 \mapsto \mathsf{Hole},\ \ldots,\ \gamma_{n-1} \mapsto \mathsf{Hole}],\ D) \\ (\texttt{Set Global}\ n\ k\ a; C,\ S,\ G,\ D) & \vdash & (C,\ S,\ G[\gamma_n \mapsto \mathsf{Fun}\ k\ a],\ D) \end{array}$$

2.3.3 Built-In Functions

The commands described in the previous section cannot by themselves completely model the domain constructs in LispLike and PowerFuL. To implement the full languages, auxiliary code sequences and new G-code instructions must be present to perform certain specific graph manipulations. Sequences of code are used when the manipulations to the graph are not simple manipulations, but instead require several steps. These code sequences are to be accessed through global heap function nodes. This section outlines the basic sequences and instructions needed for LispLike.

- To project a domain element into its respective subdomain, it is necessary to remove its embedding marker. The instruction `Retract` performs this manipulation:

$$(\texttt{Retract}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a' \vdash (C,\ a'{:}S,\ G,\ D)$$

  Then we define the sequence `_Retract-`τ as:

```
_Retract-τ:  Eval
             Retract τ
             Update 1
             Unwind
```

  We can omit the `Pop` instruction here, since `Pop 0` has no effect.

- A conditional test takes three arguments, a boolean value and two alternative results. The sequence `_If` should evaluate its first argument, and then return either its second or third argument depending on whether the first argument evaluated to **true** or **false**. This task requires the following new G-code instruction to distinguish between **true** and **false**:

$$(\texttt{Jfalse}\ C'; C,\ n{:}S,\ G,\ D),\ \ n \xrightarrow{G} \mathsf{Bool\ true} \vdash (C,\ S,\ G,\ D)$$
$$(\texttt{Jfalse}\ C'; C,\ n{:}S,\ G,\ D),\ \ n \xrightarrow{G} \mathsf{Bool\ false} \vdash (C',\ S,\ G,\ D)$$

Then the code sequence is:

```
_If:    Eval
        Jfalse _If_1
        Update 2
        Pop 1
        Unwind
_If_1:  Pop 1
        Update 1
        Unwind
```

- Deconstructing a pair involves returning either its first or its second component. While one could use higher-order functions in the ordinary λ-calculus for the same effect, it is more efficient and less complicated to use distinct tags and commands:

$$(\texttt{Left}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 \vdash (C,\ a_1{:}S,\ G,\ D)$$
$$(\texttt{Right}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 \vdash (C,\ a_2{:}S,\ G,\ D)$$

And the code sequences are:

```
_Left:  Eval              _Right:  Eval
        Left       and             Right
        Update 1                   Update 1
        Unwind                     Unwind
```

- The `Not` instruction is similar to the `Left` and `Right` instructions except that it must create a new heap node to contain its result. It cannot simply overwrite the old node since that node may be shared.

$$(\texttt{Not}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Bool}\ \beta \vdash (C,\ n'{:}S,\ G[n' \mapsto \mathsf{Bool}\ \beta'],\ D),\ \beta' = {\sim}\beta$$

and the code sequence `_Not` is almost exactly the same as for `_Left`:

```
_Not:   Eval
        Not
        Update 1
        Unwind
```

- To test whether a domain element is in a particular subdomain we check its embedding marker. The `IsEmbedded` τ commands check, for each marker, whether the indicated marker is present at the node referenced by the address on the top of the stack:

$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a' \ \vdash\ (C,\ a'{:}S,\ G[a' \mapsto \mathsf{Bool}\ \mathit{true}],\ D)$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau'\ a',\ \tau \neq \tau' \ \vdash\ (C,\ a'{:}S,\ G[a' \mapsto \mathsf{Bool}\ \mathit{false}],\ D)$$

and the associated G-code sequence,

```
_IsEmbedded-τ   Eval
                IsEmbedded τ
                Update 1
                Unwind
```

- Finally, the `_Equals` code sequence determines whether two arbitrary graph expressions are equal:

```
_Equals:  Push 0
          Eval
          Push 2
          Eval
          Equals
          Update 3
          Pop 2
          Unwind
```

The G-code instruction `Equals`, which incorporates the domain primitives **DeqD?**, **DeqA?**, **DeqB?**, **DeqF?**, **DeqP?** and **AeqA?**, is defined as follows:

$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Embed}{-}\tau\ a_1',\ a_2 \xrightarrow{G} \mathsf{Embed}{-}\tau\ a_2'$$
$$\vdash\ (\texttt{Equals}; C,\ a_1'{:}a_2'{:}S,\ G,\ D)$$
$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ a_1',\ a_2 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ a_2',\ \tau_1 \neq \tau_2$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ false}],\ D)$$
$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Atom}\ \alpha,\ a_2 \xrightarrow{G} \mathsf{Atom}\ \alpha$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ true}],\ D)$$
$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Atom}\ \alpha_1,\ a_2 \xrightarrow{G} \mathsf{Atom}\ \alpha_2,\ \alpha_1 \neq \alpha_2$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ false}],\ D)$$
$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Bool}\ \beta,\ a_2 \xrightarrow{G} \mathsf{Bool}\ \beta$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ true}],\ D)$$
$$(\texttt{Equals}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Bool}\ \beta_1,\ a_2 \xrightarrow{G} \mathsf{Bool}\ \beta_2,\ \beta_1 \neq \beta_2$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ false}],\ D)$$
$$(\texttt{Equals}; C,\ a{:}b{:}S,\ G,\ D)\ \ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2,\ b \xrightarrow{G} \mathsf{Pair}\ b_1\ b_2$$
$$\vdash\ (\texttt{Unwind},\ s{:}[\,],\ G\begin{bmatrix} s & \mapsto \mathsf{Ap}\ s'\ u \\ s' & \mapsto \mathsf{Ap}\ s''\ t \\ s'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_And} \\ t & \mapsto \mathsf{Ap}\ t'\ a_1 \\ t' & \mapsto \mathsf{Ap}\ t''\ b_1 \\ t'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_Equals} \\ u & \mapsto \mathsf{Ap}\ u'\ a_2 \\ u' & \mapsto \mathsf{Ap}\ u''\ b_2 \\ u'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_Equals} \end{bmatrix},\ \langle S, C\rangle{:}D)$$

where the G-code instruction `And` is defined:

$$(\texttt{And}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Bool\ true} \vdash (\texttt{Unwind},\ a_2{:}[\,],\ G,\ \langle S, \texttt{And}; C\rangle{:}D)$$
$$(\texttt{And}; C,\ a_1{:}a_2{:}S,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Bool\ false} \vdash (\texttt{Unwind},\ a_1{:}[\,],\ G,\ \langle S, \texttt{And}; C\rangle{:}D)$$

2.3.4 Printing

The printing command assumes that the node on top of the stack has been reduced as far as possible, *viz.*, that it has been evaluated. Printing of simple values is straightforward:

$$(O,\ \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Atom}\ \alpha \vdash (O;\alpha,\ C,\ S,\ G,\ D)$$
$$(O,\ \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Bool}\ \beta \vdash (O;\beta,\ C,\ S,\ G,\ D)$$

Printing of compounded values uses auxiliary code sequences and the `Return` command:

$$(\texttt{Return}; C, [\,], G, \langle S, C\rangle : D) \vdash (C, S, G, D)$$
$$(\texttt{Return}; C, a : [\,], G, \langle S, C\rangle : D) \vdash (C, a : S, G, D)^4$$

The rest of the definition of `Print` is then:

$$(\texttt{Print}; C, a : S, G, D),\ a \xrightarrow{G} \textsf{Embed}\text{−}\tau\ a' \vdash (\texttt{\_Print\_Embedded\_}\tau, a' : [\,], G, \langle S, C\rangle : D)$$
$$(\texttt{Print}; C, a : S, G, D),\ a \xrightarrow{G} \textsf{Pair}\ a_1\ a_2 \vdash (\texttt{\_Print\_Pair}, a_1 : a_2 : [\,], G, \langle S, C\rangle : D)$$

with code definitions

```
_Print_Embedded_τ:  Eval
                    Print
                    PrintTag τ
                    Return
```

and

```
_Print_Pair:  Eval
              Print
              Eval
              Print
              Return
```

and a trivial second printing command for domain tags,

$$(O, \texttt{PrintTag}\ \tau; C, S, G, D) \vdash (O; \tau, C, S, G, D).$$

2.4 Mapping LispLike to the Standard G-Machine

This section describes a straightforward mapping of LispLike programs onto the standard G-machine developed in §2.3. As preliminary steps of the compilation process, we first translate programs into an intermediate language, and then lambda-lift them into supercombinators. The intermediate language is exactly the notation used to describe the language semantics, except that:

- `let` bindings are not removed, since they indicate opportunities for sharing.
- `letrec` bindings are not removed, since they also indicate sharing, and since they allow a more efficient implementation of recursion than a direct implementation of the Y combinator [7].
- Since the `let` and `letrec` constructions will remain in the intermediate language, so will the identifiers associated with them.

The basic idea of the compilation scheme from the intermediate language into G-code is to generate code for each supercombinator which produces a graph expression corresponding to an expression in weak head normal from (or in head printable weak normal form, for PowerFuL expressions) denoting the same value as the supercombinator applied to the arguments on the stack. The stack configuration for evaluating a call to a supercombinator of arity n is as follows:

- The base of the stack contains a pointer to the topmost element of the spine of n application nodes which form the call to the supercombinator.
- The remaining n stack items are pointers to the n arguments to the supercombinator, with the first argument at the top of the stack.

[4] This second clause of `Return` will not be used until later, but is given here for the sake of completeness.

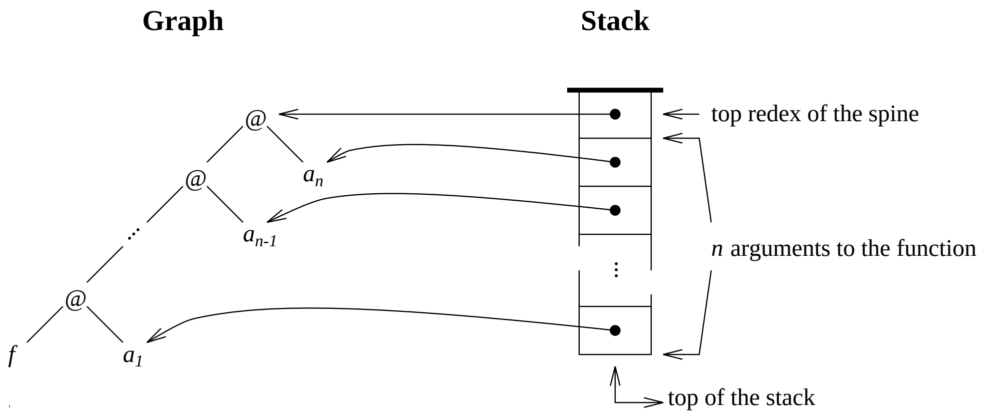


Figure 2.5: General stack configuration for a supercombinator call.

Figure 2.5 shows this configuration. To produce the desired graph expression, the code generated for each supercombinator will then

1. construct a graph corresponding to the expression denoted by the supercombinator, substituting the actual parameters in the stack for the formal parameters in the supercombinator definition,
2. update the graph node corresponding to the top of the spine of the application of the supercombinator to the given arguments with the newly constructed graph,
3. remove the actual parameters from the stack, and
4. reduce the expression to weak head-normal form.

By expressing the overall, top level of the program as a single distinguished supercombinator, we transform the task of executing the program into the task of evaluating this supercombinator.

2.4.1 Translation Schemes

From this outline, we can derive a translation scheme $\mathcal{SC}$ to convert supercombinators into G-code:

$$\begin{aligned}
&\mathcal{SC}\llbracket \mathbf{\$S}\ x_1\ \ldots x_n\ =\ E \rrbracket \\
&\quad = \ \texttt{Glob Start } \mathbf{\$S}\ n; \mathcal{C}\llbracket E \rrbracket\, [x_1 \mapsto n,\ \ldots,\ x_n \mapsto 1]\, n; \texttt{Update } n; \texttt{Pop } (n-1); \texttt{Unwind}
\end{aligned}$$

where $\mathcal{C}$ constructs the graph corresponding to an expression. Then to compile an entire program, consisting of a list of supercombinator definitions where the "main" supercombinator is listed first, we can use $\mathcal{PROG}$ scheme:

$$\begin{aligned}
&\mathcal{PROG}\llbracket \mathbf{\$S}_0\ x_{00}\ \ldots x_{0r_0} = E_0; \ldots; \mathbf{\$S}_n\ x_{n0}\ \ldots x_{nr_n} = E_n \rrbracket \\
&\quad = \ \texttt{Alloc Globals } (n+1); \texttt{Set Global } 0\ r_0\ \Gamma_0; \ldots; \texttt{Set Global } n\ r_n\ \Gamma_n; \\
&\qquad \texttt{Push Global } 0; \texttt{Eval}; \texttt{Print}; \texttt{Halt}; \\
&\qquad \Gamma_0 : \mathcal{SC}\llbracket \mathbf{\$S}_n\ x_{00}\ \ldots x_{0r_0} = E_0 \rrbracket; \\
&\qquad \vdots \\
&\qquad \Gamma_n : \mathcal{SC}\llbracket \mathbf{\$S}_n\ x_{n0}\ \ldots x_{nr_n} = E_n \rrbracket;
\end{aligned}$$

We define the $\mathcal{C}$ scheme in terms of each type of expression:

$$\boxed{\mathcal{SC}[\![\mathbf{\$S}\ x\ y\ z = (z = \mathbf{A}_5)]\!] = \texttt{Push Atom } 5; \texttt{Push } 3; \ldots}$$

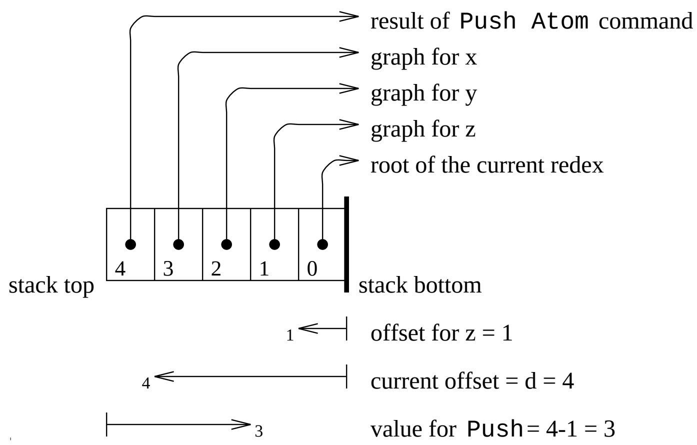


Figure 2.6: Calculation of the offset for the `Push` command.

- Simple scalar constants and global values can be simply created and pushed onto the stack.

$$\begin{aligned}
\mathcal{C}[\![\mathbf{A}_n]\!]\,\rho\,d &= \texttt{Push Atom } n \\
\mathcal{C}[\![\beta]\!]\,\rho\,d,\ \beta \text{ a boolean value} &= \texttt{Push Bool } n \\
\mathcal{C}[\![\mathbf{\$S}]\!]\,\rho\,d &= \texttt{Push Global } (\rho\ \mathbf{\$S})
\end{aligned}$$

- A pointer to an actual parameter, or to a bound value, can be copied from its slot within the stack when it was first pushed. The distance from the value's slot to the *bottom* of the stack is stored in the environment, so the offset from the current *top* of the stack, which the `Push` command needs (p. 16), is the current depth minus the depth of the slot, as illustrated in Figure 2.6. The rule is then:

$$\mathcal{C}[\![x]\!]\,\rho\,d,\ x \text{ a bound name} = \texttt{Push } (d - \rho\,x)$$

- Given the graphs of two expressions, constructing the graph of the ordered pair containing those two expressions is simply building a Pair node which references the two expressions' graphs. Similarly, constructing an embedded value consists of simply building the appropriate Embed-τ node with a reference to the proper graph.

$$\begin{aligned}
\mathcal{C}[\![\langle e_1, e_2\rangle]\!]\,\rho\,d &= \mathcal{C}[\![e_2]\!]\,\rho\,d; \mathcal{C}[\![e_1]\!]\,\rho\,(d+1); \texttt{MkPair} \\
\mathcal{C}[\![e_\tau]\!]\,\rho\,d &= \mathcal{C}[\![e]\!]\,\rho\,d; \texttt{MkEmbed } \tau
\end{aligned}$$

- To construct the graph of an application, first the argument and then the function are pushed onto the stack, and then an application node referring to the two is created.

$$\mathcal{C}[\![e_1\ e_2]\!]\,\rho\,d = \mathcal{C}[\![e_2]\!]\,\rho\,d; \mathcal{C}[\![e_1]\!]\,\rho\,(d+1); \texttt{MkAp } 1$$

- For a non-recursive binding, it is a simple matter to construct the bound value on the graph. The offset of the graph from the bottom of the stack is stored, and the graph of the overall expression first constructed and then slid over to the bound value's slot.

$$\mathcal{C}[\![\textbf{let } id \textbf{ be } val \textbf{ in } expr]\!]\,\rho\, d \;=\; \mathcal{C}[\![val]\!]\,\rho\, d; \mathcal{C}[\![expr]\!]\,\rho[id \mapsto d+1]\,(d+1); \texttt{Slide } 1$$

- Recursive bindings are more complex because the new environment must be fully constructed before any of the graphs are built, and we will introduce two new translation schema to break down the process into simpler steps. We will first call the $\mathcal{X}R$ scheme to construct the new environment. We next reserve space on the stack and graph for the bound values so that each may refer to the areas in the graph where the graphs of the other expressions *will be* before reduction of the overall expression begins. The $\mathcal{C}L$ scheme will then fill these reserved slots with the graphs of the bound values. Finally the graph of the overall expression is constructed and then moved down the stack to overwrite the bound values' slots.

$$\begin{aligned}\mathcal{C}[\![\textbf{letrec } \boldsymbol{decls} \textbf{ is } \boldsymbol{expr}]\!]\,\rho\, d \;&=\; \mathcal{C}L[\![decls]\!]\,\rho'\, d'; \mathcal{C}[\![expr]\!]\,\rho'\, d'; \texttt{Slide } d'-d \\ &\quad\; \text{where } (\rho',\, d') = \mathcal{X}R[\![decls]\!]\,\rho\, d\end{aligned}$$

  The $\mathcal{X}R$ and $\mathcal{C}L$ schemes are defined below.

- The remaining translation rules all follow the same basic pattern: the arguments (subexpressions) are pushed onto the stack in reverse order and followed by a global function value, then an application spine is created from these values.

$$\begin{aligned}
\mathcal{C}[\![e|_\tau]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e]\!]\,\rho\, d; \texttt{Push Global } \$\textsc{Retract-}\tau; \texttt{MkAp } 1\\
\mathcal{C}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e_3]\!]\,\rho\, d; \mathcal{C}[\![e_2]\!]\,\rho\,(d+1); \mathcal{C}[\![e_1]\!]\,\rho\,(d+2);\\
&\quad\;\; \texttt{Push Global } \$\textsc{If}; \texttt{MkAp } 3\\
\mathcal{C}[\![\textbf{left } e]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e]\!]\,\rho\, d; \texttt{Push Global } \$\textsc{Left}; \texttt{MkAp } 1\\
\mathcal{C}[\![\textbf{right } e]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e]\!]\,\rho\, d; \texttt{Push Global } \$\textsc{Right}; \texttt{MkAp } 1\\
\mathcal{C}[\![\textbf{not } e]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e]\!]\,\rho\, d; \texttt{Push Global } \$\textsc{Not}; \texttt{MkAp } 1\\
\mathcal{C}[\![e \in \tau]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e]\!]\,\rho\, d; \texttt{Push Global } \$\textsc{IsEmbedded-}\tau; \texttt{MkAp } 1\\
\mathcal{C}[\![e_1 = e_2]\!]\,\rho\, d \;&=\; \mathcal{C}[\![e_2]\!]\,\rho\, d; \mathcal{C}[\![e_1]\!]\,\rho\,(d+1);\\
&\quad\; \texttt{Push Global } \$\textsc{Equals}; \texttt{MkAp } 2
\end{aligned}$$

The $\mathcal{X}R$ scheme builds a new environment and a new offset counter for binding mutually recursive definitions.

$$\mathcal{X}R[\![x_1 = E_1,\, \ldots,\, x_n = E_n]\!]\,\rho\, d = (\rho[x_1 \mapsto d+1,\, \ldots,\, x_n \mapsto d+n],\, d+n)$$

The $\mathcal{C}R$ scheme creates G-code to instantiate the bound values on the stack.

$$\begin{aligned}\mathcal{C}L[\![x_1 = E_1,\, \ldots,\, x_n = E_n]\!]\,\rho\, d = \texttt{Alloc } n;\ & \mathcal{C}[\![E_1]\!]\,\rho\, d; \texttt{Update } n;\\ & \quad\vdots \\ & \mathcal{C}[\![E_n]\!]\,\rho\, d; \texttt{Update } 1\end{aligned}$$

2.4.2 Standard Global Values

The final section of rules given for the $\mathcal{C}$ translation scheme referred to global values not previously defined, such as $If and $Left. A final stage of the process of translation into G-code must integrate code to set up these global references into the final G-code program. Specifically, this linking phase must

1. determine which of these global values are accessed,
2. attach the relevant G-code sequences to the compiled code,

3. add additional `Set Global` instructions to set up graph nodes for these globals, and
4. replace the labels of the standard globals with their actual global position names.

These globals, and the functions and arities associated with them, are as follows:

Global Name	Code Sequence	Arity
$RETRACT-$\tau$	_Retract-τ	1
$IF	_If	3
$LEFT	_Left	1
$RIGHT	_Right	1
$NOT	_Not	1
$ISEMBEDDED-$\tau$	_IsEmbedded-τ	1
$EQUALS	_Equals	2

Thus, for example, when the initial translation of G-code contains a reference to the global $NOT, during the global linking the program must be altered as follows:

1. The space reserved for globals must be increased by one,
2. The code sequence labeled `_Not` must be appended to the program,
3. A new instruction must be added to the list of `Set Global` commands, moving a reference to the location of the code sequence `_Not` and an arity of one into the new global value, and
4. All references to $NOT must be replaced with the actual number of the new global slot.

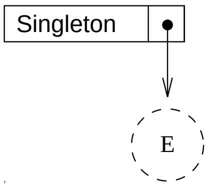
Chapter 3

A Graph Representation for PowerFuL Programs

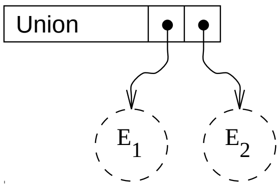
To motivate an abstract machine for PowerFuL, we consider the role of logic variables in a graph reduction-based implementation of the language. As for many graph models, we consider an intermediate graph "language," but unlike most models, the intermediate language will very closely resemble the optimized denotational semantics of PowerFuL. We make this design decision first because the denotational semantics for PowerFuL already closely resemble the sort of intermediate language used for graph implementations, such as an enriched lambda calculus [7], and secondly because it is only in the semantics that logic variables appear. We do not use the semantics themselves because of the inefficiency of the Y combinator in a graph environment. Expressions which do not involve logic variables reduce in the ordinary manner. It is the reduction of expressions over logic variables, and the associated manipulations of the graph, which we treat here.

3.1 Representing Basic Set Constructs

We will represent the basic set constructs in the graph in a manner similar to that of basic data. The empty set is represented by a single node with no children; distinct from singleton sets, modeled by a node tagged as a singleton, with one child for the value of the item in the set:



Unions are modeled by a node tagged as a union node, with two children for the sets to be united:



When it simplifies the graphic renderings, we will abbreviate the union nodes as

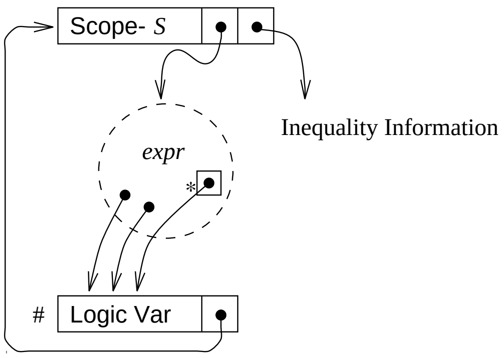
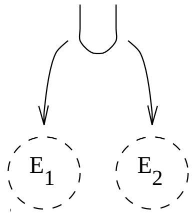
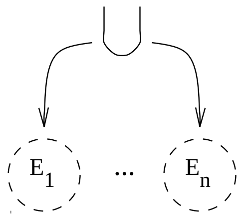


Figure 3.1: The graph representation of an expression of the form $set(X).expr$.



This notation lends itself well to expressing arbitrary finite union, *viz.*,



Function distribution over a set ($^+$), rather than being treated with a distinct node type, is simply a possible content of a function node.

3.2 Representing Logic Variables

Each occurances of a logic variable in expressions denoted by PowerFuL programs has a clearly defined scope: A logic variable is part of the description of some set of values, and is always introduced along with the set of values over which that logic variable may range. For example, the expression

$$atoms(X).expr$$

declares that the logic variable X may range over the set of atomic values in the subexpression $expr$. In a graph representation of such an expression, the graph node corresponding to the declaration will contain all information about that logic variable. This declaration will also mark the scope of the variable. All references to the logic variable itself can also be placed in a single graph node, which simply refers back to the declaration node. Figure 3.1 shows the graph representation of an expression of the form

$$S(X).expr,$$

where G is the graph representation of a value of the form $expr(X)$, and where X refers to a unique representation of a logic variable whose scope is marked by the Scope node. The "inequality information" refers to constraints on specific values to which the quantifier may never be bound, and is discussed in §3.4.

A logic variable will always take the value of a member of the Herbrand Universe, so in a sense it is in normal form. However, it will often be the case that an expression which would be a redex if one of its subexpressions were in (weak-head) normal form will not be immediately reducible if that subexpression were a logic variable. For example, the expression

$$(\text{'}\mathbf{foo})_a \in a$$

is a redex, and would reduce to **true**, but if $\text{'}\mathbf{foo}_a$ were replaced by a logic variable ranging over the Herbrand universe, no reduction could be made without some manipulation of that variable. Since expressions such as

$$\bigcup_{x \in \mathbf{hu}} \{\langle x, \mathbf{A}_{2a}\rangle_p\}$$

should be taken as in normal form, it is necessary that a logic variable by itself be taken as in normal form. A *potential* redex is defined to be an expression that can be made into a redex by replacing some or all of the logic variables contained as its proper subexpressions[1] with ground values. A redex is always a potential redex, but the reverse is not always true. An expression which is a potential redex but not a redex is defined to be *impeding*.

Thus, an abstract machine for PowerFuL will need to look for the outermost potential redex, rather than the outermost redex. In the figures, the expression currently being considered as the outermost potential redex will be marked '#', and the expression previously considered will be marked '*'. When the machine encounters an impeding potential redex, it first checks the surrounding environment for the declaration of the logic variable: the reduction may become immediately reducible based on the inequality information stored there. If this is not the case, the logic variable must be narrowed into more than one case of smaller range.

3.3 Reducing Logic Variables

The steps for reduction of a logic variable node are as follows: when the traversal of the graph through outermost redexes arrives at a logic variable, we examine the context of this use of the variable. The context will determine what information will be needed, and the availability or lack of this information will determine whether narrowing will be necessary. Figure 3.2 shows a generalization of the manipulation of the graph in circumstances where we have sufficient information to reduce the next-to-outermost redex.

3.3.1 Narrowing and Quasishared Nodes

The manipulation of expressions produced by the optimized denotational semantics of PowerFuL may involve the transformation of a set generated by a logic variable ranging over some other set into the union of sets each generated by logic variables ranging over sets which may or may not be the same as the original generating set. That is, an expression of the form

$$set(X).expr(X)$$

reduces to one of the form

$$\bigcup_i \left(set_1^i(X_1^i).set_2^i(X_2^i).\ldots.set_{n_i}^i(X_{n_i}^i).expr\left(\psi_i(X_1^i, X_2^i, \ldots, X_{n_i}^i)\right)\right). \tag{3.1}$$

[1] It would not be correct to simply call for *any* subexpression here. Since any expression is a subexpression of itself, if we did not restrict this definition to proper subexpressions then a logic variable by itself or an expression such as the one above would be taken as a potential redex, when if fact it should be taken as in full normal form.

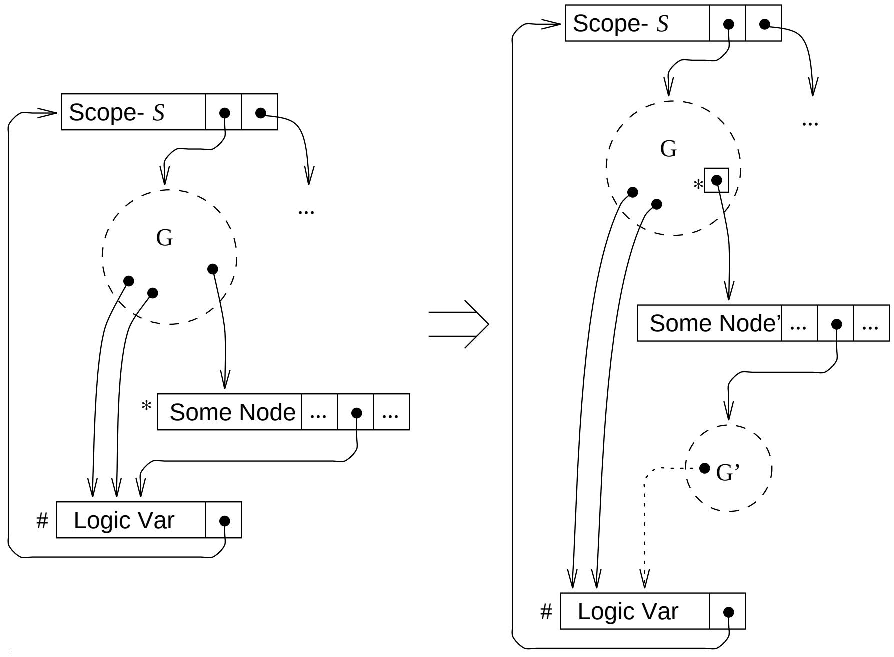


Figure 3.2: The general manipulation of a graph which corresponds to narrowing being unnecessary, and the subsequent reduction of the enclosing expression. The dotted line from the subgraph G' indicates that the corresponding subexpression may or may not contain additional occurances of the original logic variable.

For an expression $\bigcup_{x \in S}(\ldots(f\ x)\ldots)$, where $(f\ x)$ is the outermost potential redex, and given S and f:

S	f	n	sets	ψ_i's
hu	`Retract` τ `IsEmbedded` τ $=$ Embed$-\tau$ e	3	$set_1^1 = \mathbf{atoms}$, $n_1 = 1$ $set_1^2 = \mathbf{bools}$, $n_2 = 1$ $set_1^3 = set_2^3 = \mathbf{hu}$, $n_3 = 2$	$\psi_1(x) = x_a$ $\psi_2(x) = x_b$ $\psi_3(x,y) = \langle x, y\rangle_p$
	$=$ Logic Var ℓ	2	$n_1 = 0$ $set_1^2 = \mathbf{hu} \setminus \{\ell\}$, $n_2 = 1$	$\psi_1 =$ Logic Var ℓ $\psi_2(x) = x$

bools	`If`
	`Not`
	$=$
		2	$n_1 = n_2 = 0$	$\psi_1 = \mathbf{true}$
				$\psi_2 = \mathbf{false}$

atoms	$= \mathbf{A}_k$	2	$n_1 = 0$ $set_1^2 = \mathbf{atoms} \setminus \{\mathbf{A}_k\}$, $n_2 = 1$	$\psi_1 = \mathbf{A}_k$ $\psi_2(x) = x$
	$=$ Logic Var ℓ	2	$n_1 = 0$ $set_1^2 = \mathbf{atoms} \setminus \{$Logic Var $\ell\}$, $n_2 = 1$	$\psi_1 =$ Logic Var ℓ $\psi_2(x) = x$

where $\tau \in \{a, b, p\}$.

Table 3.1: Narrowing manipulations, in the notation of equation (2.1).

For example, two rules for simplifying the optimized denotational semantics of PowerFuL programs are

$$\bigcup_{X \in \mathbf{hu}} E_1 \implies \bigcup_{Y \in \mathbf{atoms}} \left(E_1 \left[\frac{Y_a}{X}\right]\right) \cup \bigcup_{Y \in \mathbf{bools}} \left(E_1 \left[\frac{Y_b}{X}\right]\right) \cup \bigcup_{Y \in \mathbf{hu}} \bigcup_{Z \in \mathbf{hu}} \left(E_1 \left[\frac{\langle Y, Z\rangle_p}{X}\right]\right)$$

and

$$\bigcup_{X \in \mathbf{bools}} E_2 \implies E_2 \left[\frac{\mathbf{true}}{X}\right] \cup E_2 \left[\frac{\mathbf{false}}{X}\right].$$

Expressing these manipulations in the notation of equation 3.1, we have for the former

$$set_1^1 = \mathbf{atoms},\ n_1 = 1,\ \psi_1(x) = x_a,$$
$$set_1^2 = \mathbf{bools},\ n_2 = 1,\ \psi_2(x) = x_b,$$
$$set_1^3 = set_2^3 = \mathbf{hu},\ n_3 = 2,\ \psi_3(x,y) = \langle x, y\rangle_p$$

and for the latter

$$n_1 = n_2 = 0,\ \psi_1 = \mathbf{true},\ \psi_2 = \mathbf{false}.$$

Tables 3.1 and 3.2 list the various cases of narrowing and reduction, respectively, in the context of logic variables.

The graph transformation corresponding to this sort of reduction is referred to as *narrowing* [9]. Narrowing will, in general, produce more than one subgraph corresponding to the original narrowed expression. It would be inefficient to duplicate the entire original subgraph, so we can use graph nodes which indicate areas of graph shared by different expressions produced in narrowing.

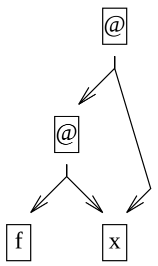
For an expression $\bigcup_{x \in S}(\ldots(f\ x)\ldots)$ where $(f\ x)$ is the outermost potential redex, given S and f, to find the u such that

$$f\ x \longrightarrow u.$$

S	f	u
hu$\backslash${Logic Var ℓ}	= Logic Var ℓ	**false**
atoms$\backslash\{\mathbf{A}_i\}$	$= \mathbf{A}_i$	**false**
atoms$\backslash${Logic Var ℓ}	= Logic Var ℓ	**false**

Table 3.2: Situations involving logic variables which allow reduction rather than narrowing.

Note that this sharing is not quite the same as the usual sense of "sharing" in a graph: usually entire subexpressions are shared, such as in the representation of expressions like `let x = ... in (f x x)`:



In the manipulation corresponding to the simplification of a logic variable, however, only part of an expression is shared. To distinguish between the two notions of sharing, fragments of the graph which are shared between subsets, but which do not represent complete subexpressions themselves, will be refered to as *quasishared* graph elements.

Instead of decorating the graph of the program, we could also handle the situation by β-abstracting the non-shared expressions so that the resulting shared function can be treated as such, with the overall subexpression as the union of applications of this new function. While such an approach may not be inappropriate for graphical or interpretive environments, in a compiled environment a run-time generation of new β-abstractions will require the compilation and linking of new code during program execution. In a program which makes wide use of set generators ranging over the Herbrand Universe, this code generation could become quite a burden, and we do not consider this method further.

Just as we can generalize this sort of manipulation of expressions as in (3.1), we can express the analogous manipulation of the graph in general terms, as shown in Figure 3.3[2]. The figure shows the transformation applied to a graph when the logic variable in node β becomes the outermost redex. The graph manipulator will generate n cases based on the context of the logic variable and the set S over which it ranges. The subgraphs I_i correspond to the declarations for the logic variables X^i_j and the sets set^i_j over which the logic variables range; each J_i corresponds to the expression $\psi_i(X^i_1, X^i_2, \ldots, X^i_{n_i})$, that is, the value with which the original logic variable is replaced in a particular subexpression. To delimit the areas of the graph which are quasishared, nodes tagged Rejoin-n and Split-n, where n is the number of subgraphs produced, are generated at the top and bottom, respectively, of the quasishared fragment.

[2]The Ex-Scope node displayed in the figure has no meaning in the graph model, and could be treated as indirection nodes, and removed when convenient. However, these nodes will be used in the actual abstract machine.

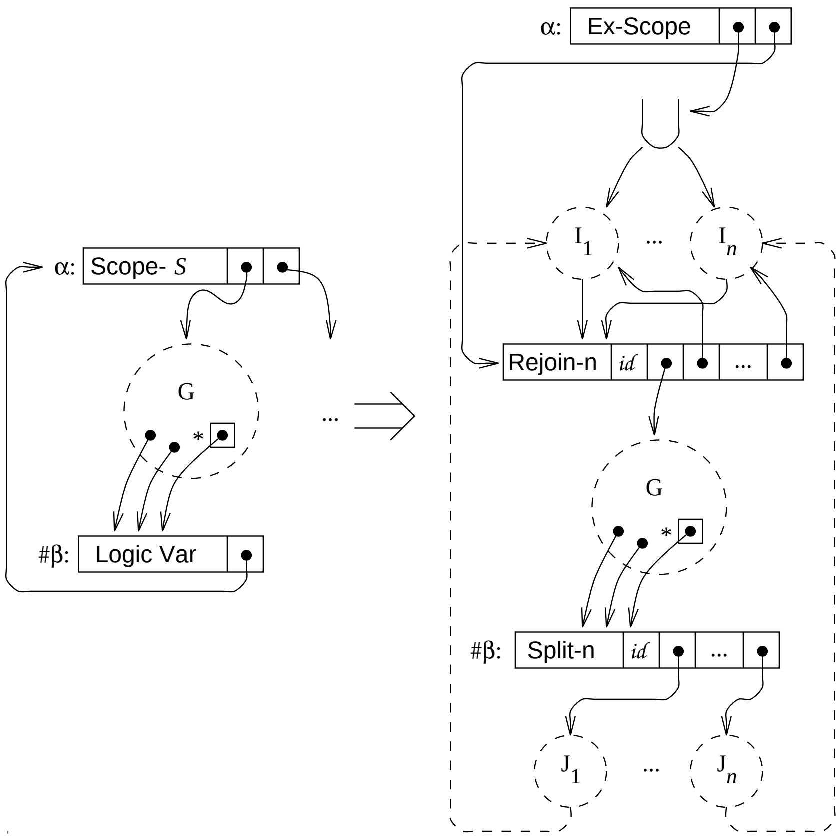


Figure 3.3: The general manipulation of a graph which corresponds to narrowing being necessary to acquire more information about an unbound logic variable which has become the outermost redex.

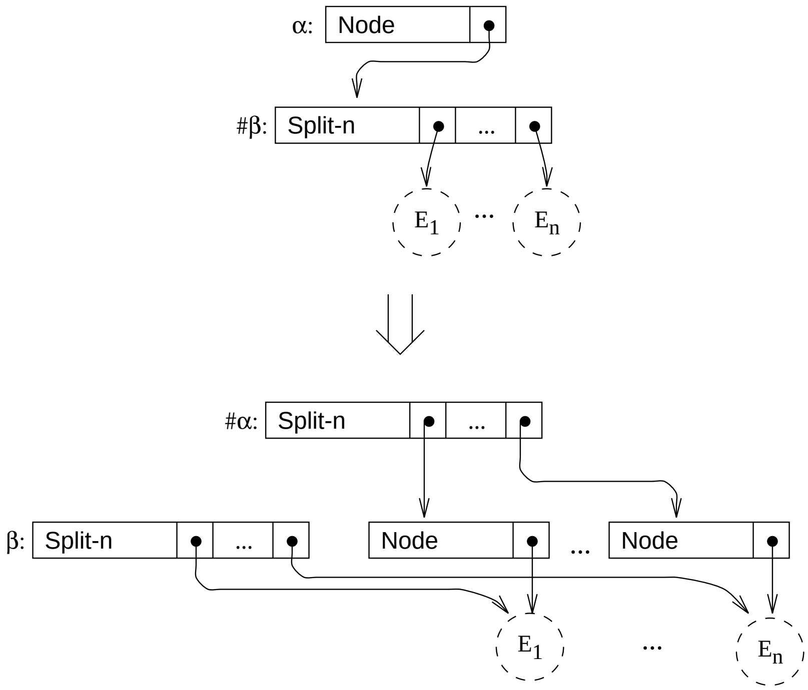


Figure 3.4: The simplest case of upward propagation of the Split node, with no other non-splitting arguments in the enclosing graph node.

3.3.2 Reducing and Eliminating Quasisharing Delimiters

The previous paragraphs outline a representation of non-ground quantifiers in a program's graph, and presented a scheme to allow graph elements of different subexpressions formed in the narrowing of non-ground quantifiers to be shared. This section describes the role of these new structures in the reduction of the graph and the manipulations to be performed based on their presence.

Upward Propagation of Split-n Nodes

Reduction of the Split-n node consists of propagating the Split upwards through the graph, thus converting each single quasishared graph node immediately above the Split into n separate nodes contained beneath a new Split node. When the graph is traversed, those subgraphs which correspond to the subexpressions in which an enclosing expression is strict are those which will be selected for reduction. Since these redexes may be transformed into a different expression for each of the cases generated by the narrowing, the nodes corresponding to the tops of these redexes cannot be quasishared. When the node enclosing a Split-n tagged node is not a Rejoin-n tagged node, the reduction is fairly straightforward. If the enclosing node points to no other arguments, then the transformation is trivial, as shown in Figure 3.4. It is somewhat more complicated to perform the manipulation when the Split-n node is one of several arguments. The subgraphs corresponding to these other arguments may or may not contain occurances of Split-n nodes. Since they may, and since an exhaustive search of the subgraphs would be prohibitively costly, it must be assumed that those subgraphs are quasishared, and therefore must be enclosed under Rejoin-n tags. Figure 3.5 shows a general description of this sort of manipulation. The situation where there are no other arguments is clearly a special case of this general scheme: with respect to the notation of Figure 3.5, we simply take $k = 0$.

When a Split-n node is propagated all the way to the corresponding Rejoin-n node, both of

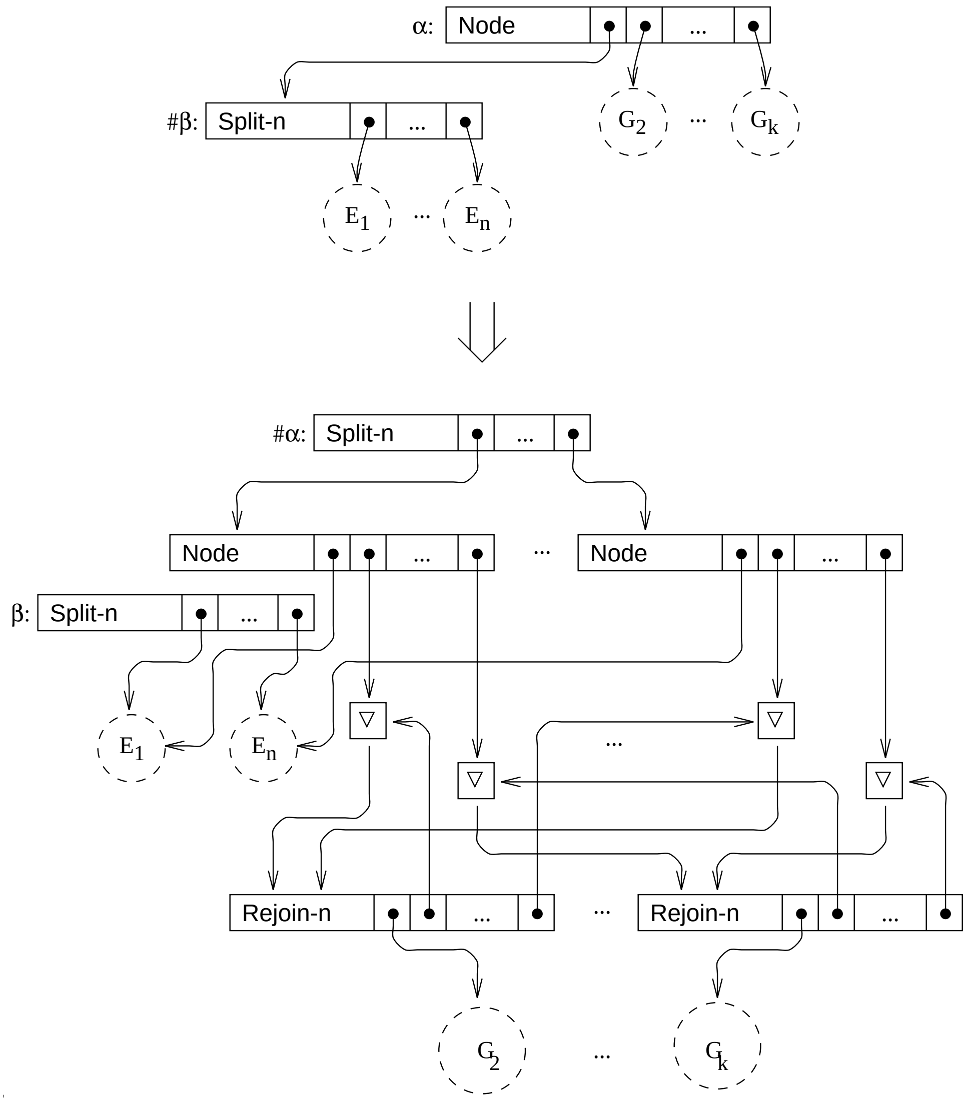


Figure 3.5: Upward propagation of the Split node where other non-splitting arguments are present in the enclosing node.

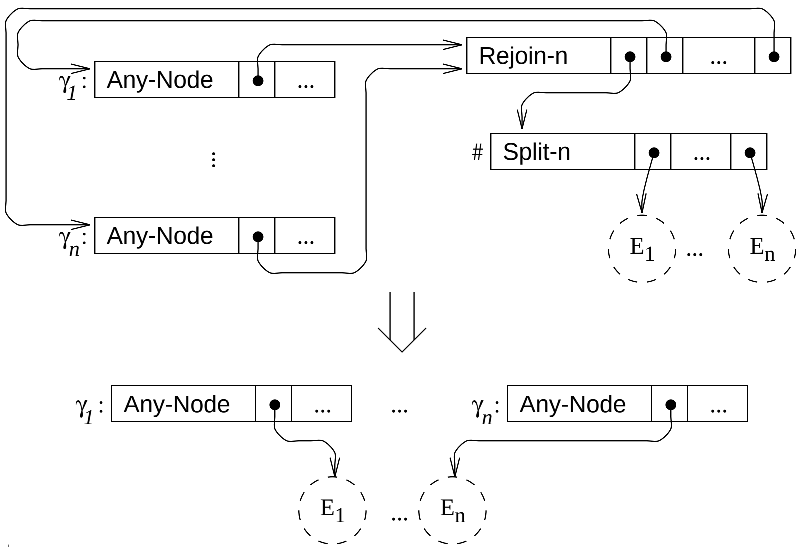


Figure 3.6: Elimination of a Split-n/Rejoin-n pair. Note that the next redex is not specified in the figure: the redex active just prior to the current redex will again be the current redex, since it would be difficult to determine which of the n nodes called the shared instance of the Split node.

these nodes are removed from the graph. The n nodes which refer to the Rejoin-n node will be altered to refer instead to the n nodes referred to by the Split-n node: that is, the i^{th} node pointing to the Rejoin-n node will instead point to the i^{th} child of the Split-n node. Figure 3.6 shows this manipulation of the graph. It is to facilitate this manipulation that we have required that Rejoin nodes contain a pointer to any node pointing back to it; in §4.5.2 we further refine the treatment of Rejoin nodes. The node of the graph which originally immediately enclosed the node onto which the Rejoin-n node in question was written (before the narrowing which produced that Rejoin-n node) will be the node with which the machine resumes its search for the current redex.

Example

To illustrate the motion of a Split-n node through the graph immediately after narrowing, we step through the evaluation of the expression

$$\bigcup_{x \in \mathbf{bool}} \{\mathbf{if}\ x\ \mathbf{then\ true\ else\ false}\}$$

Figure 3.7 shows the state of the graph just before the narrowing over x. Forcing the evaluation of the If node causes evaluation of its first subexpression, in which the corresponding domain predicate **if** is strict. This subexpression whose value we must test as either true or false is a logic variable; therefore the range of that variable narrows to the individual values of its domain **bool**. Figure 3.8 shows the state of the graph immediately after the narrowing. Using the notation of equation 3.1 and Figure 3.3, we have $n = 2$, $I_1 = I_2 = \nabla$, the indirection node, $J_1 =$ True and $J_2 =$ False, both J_i's a single node.

Once the graph is narrowed, the previous redex becomes a Split node rather than a node for a logic variable. Using the rules above, the Split node propagates upwards through the graph towards its corresponding Rejoin node. The node moves upward twice: the first step moves it above the If node; the second moves it above the Singleton node. For the first manipulation, the presence of other subexpressions means that the scheme of Figure 3.5 applies, while for the latter the simpler scheme of Figure 3.4 is sufficient. These two manipulations are illustrated in

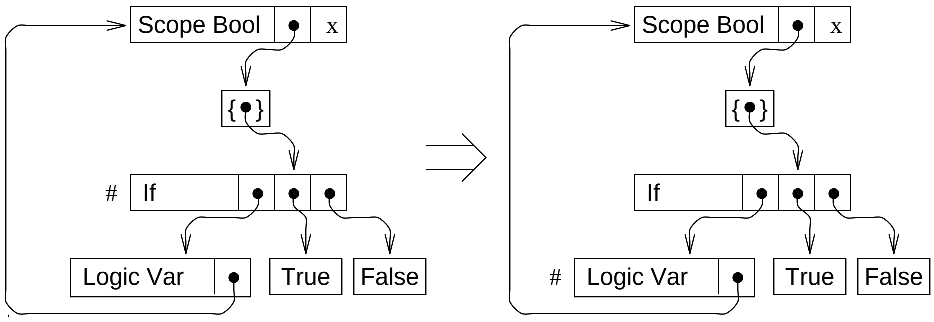


Figure 3.7: State of the graph immediately before narrowing the logic variable in the expression $\bigcup_{x \in \mathbf{bool}} \{\mathbf{if}\ x\ \mathbf{then\ true\ else\ false}\}$.

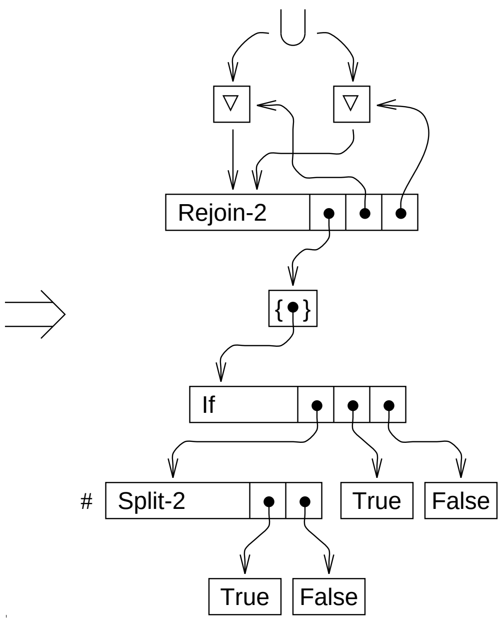


Figure 3.8: The result of narrowing in the example above.

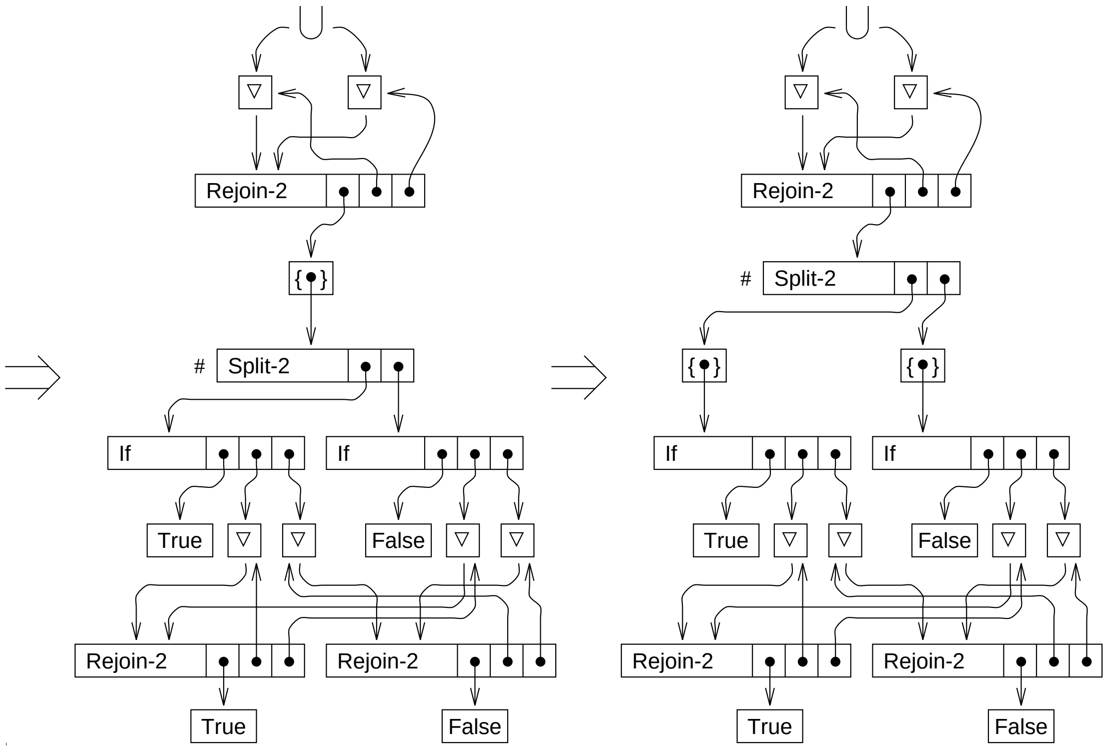


Figure 3.9: Upward propagation of the Split node in the example.

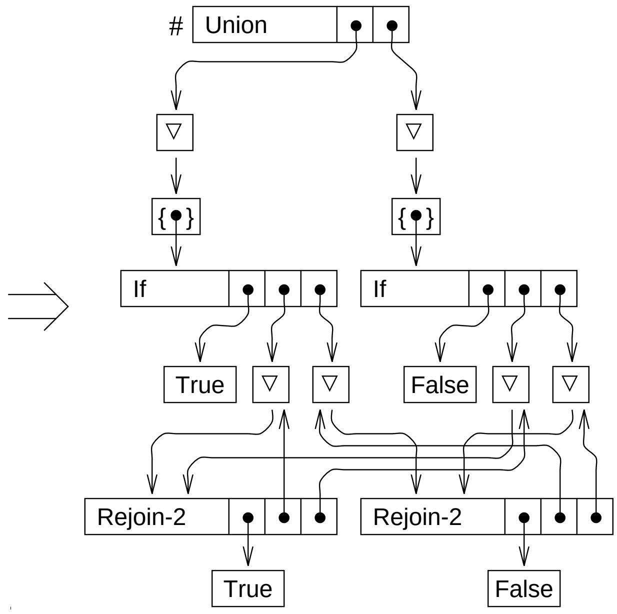


Figure 3.10: Elimination of a pair of Split and Rejoin nodes.

Figure 3.9. After these two steps the Split node will be just below the Rejoin node, so the two are removed according to the scheme of Figure 3.6; this manipulation in the example is shown in Figure 3.10.

Downward Propagation of Rejoin-n nodes

The example above raises a new issue: what should happen when the search for the outermost redex encounters a Rejoin node? Clearly this situation is possible: consider the evaluation of the expression

$$f^{+}S$$

where $f : \mathtt{B} \rightarrow \mathcal{P}(\mathtt{X})$, f strict, that is, f is any function from boolean values into any collection of sets which forces its argument to be computed at least to some head-normal form, and S is the expression in the previous example. Evaluating S we quickly arrive at a Rejoin node, as shown in Figure 3.11.[3] In the figure, first the If node is evaluated: its first argument is a True node, so the node is updated with an indirection to its second argument, *as per*

$$\mathbf{if(TRUE,}\ y,\ z) = y.$$

That new indirection node is still the outermost redex, so control flows through that indirection (as indicated by the dotted line), and through a second one below it, to the Rejoin node.

The simplest rule for encountering Rejoin nodes is to treat them analogeously to Split nodes. Since Split and Rejoin delimit areas of shared nodes in the graph, the behavior of a Rejoin node could be simply to "unshare" the node immediately below it, and update the leftmost slots of the nodes known to point to it with pointers to copies of the newly unshared node. Figure 3.12 depicts this manipulation. Because it may not be possible to immediately tell exactly which of the resulting n new nodes correspond to what should be the current redex, control would return to the node previously considered to be the outermost redex, and would then flow immediately

[3]Note that extranneous indirection nodes have been removed from the depiction of the graph.

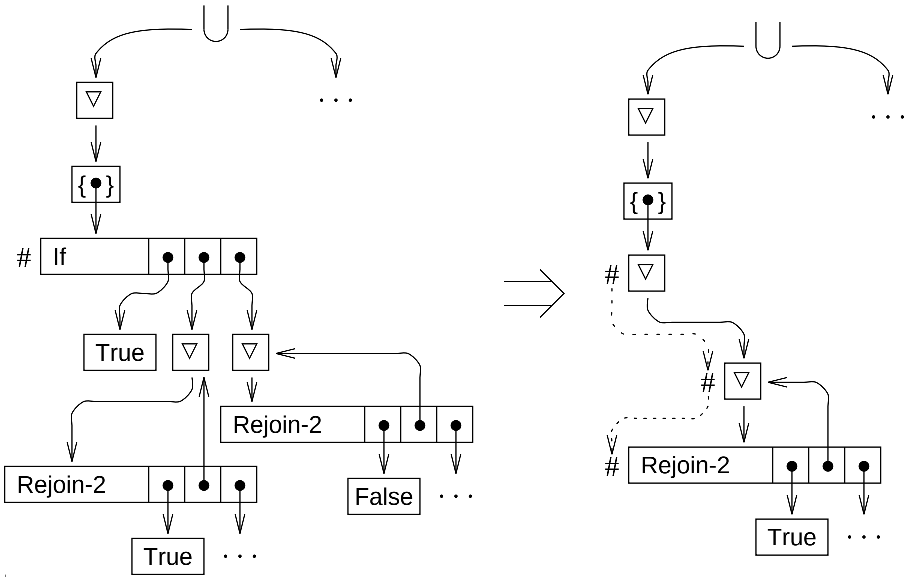


Figure 3.11: Derivation of a Rejoin node as the outermost redex.

to its new subexpression just as it flowed to the Rejoin node as before. In the case where $m = 0$, that is, in the case where the node just below the Rejoin node has no children, the Rejoin node would disappear from the graph as there would no longer be any shared graph nodes below it. This simple case is exactly the occurance which would conclude the running example, as shown in Figure 3.13.

But the approach above is extravagant because the view that Split and Rejoin nodes should be treated similarly is flawed. Encountering a Split node does indicate that quasishared nodes must be duplicated and unshared, because such an encounter indicates that the outermost redex is not shared. However, the outcome of the running example showed that encountering a Rejoin node does not guarantee an encounter with a Split node, and moreover, that arbitrary downward propagation of Rejoin nodes may waste graph space. Of course, in the example only a negligible amount of space was wasted, but a program could easily have a static, fully grounded data structure of considerable size in such a situation.

On the opposite extreme from *always* propagating Rejoin nodes is *never* propagating them. That is, one could adopt as a rule for encountering Rejoin nodes to treat them almost like indirection nodes, and ignore them except when moving a Split node upwards. This rule simply will not work. In fact, a sufficient guideline is that the Rejoin node should be propagated only when some manipulation requires access to the subgraphs of the node under the Rejoin. The reason for this guideline will be explained in §4.5.4, p. 77.

Logic Variables Over Unions

The reduction axiom for the declaration of a logic variable whose contained subexpression is a union of two sets requires some special attention. The axiom is:

$$set(X).(E_1 \cup E_2) = (set(X_1).E_1) \cup (set(X_2).E_2)$$

Even though the two logic variables, at the time of narrowing, will range over the same set, it would be incorrect to use the same scoping declaration for both. Subsequent graph reduction may require narrowing for one but not for the other, or for each in different ways, and it would be inappropriate to narrow the other unnecessarily. However, when the graph of such an expression

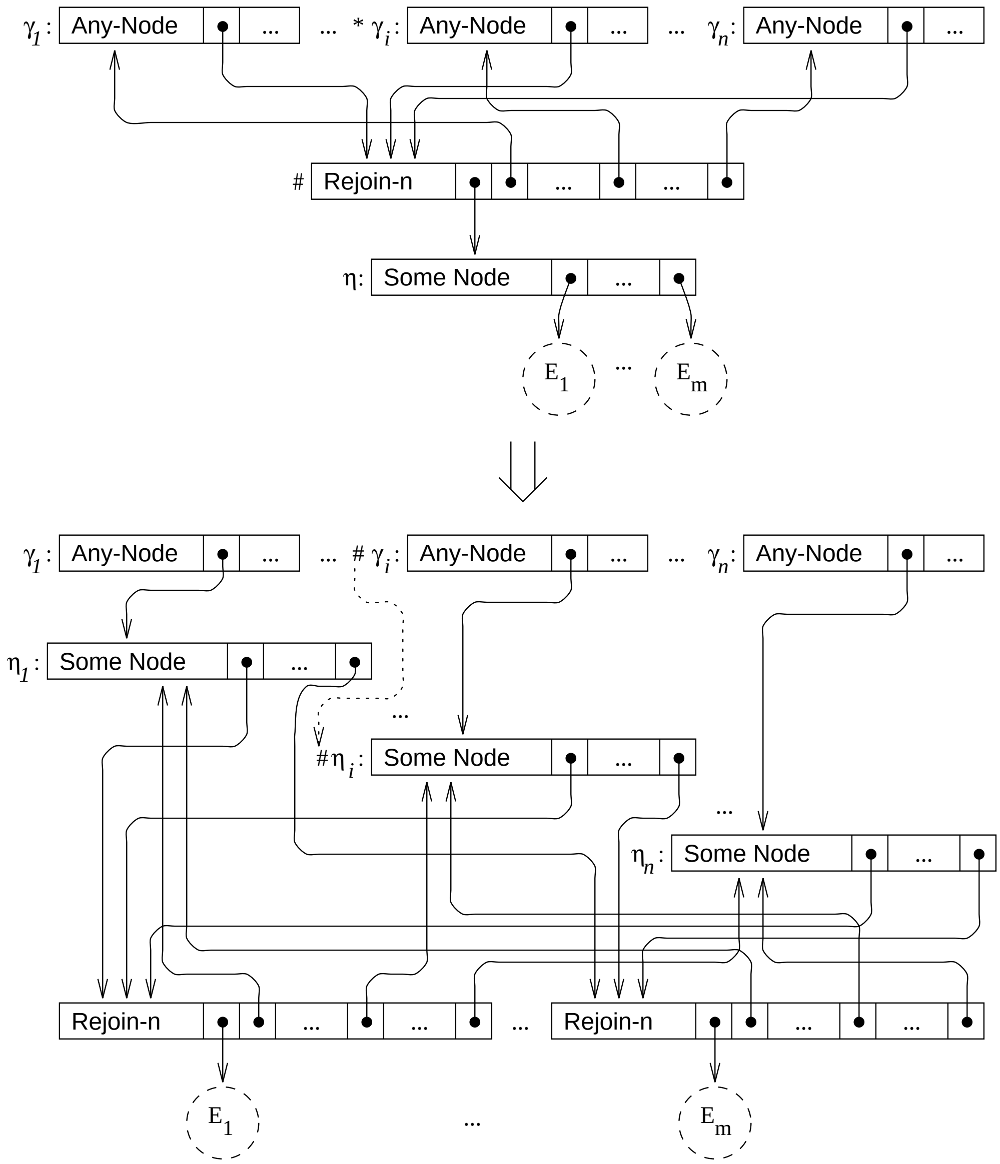


Figure 3.12: A naïve scheme for dealing with Rejoin nodes as the outermost redex.

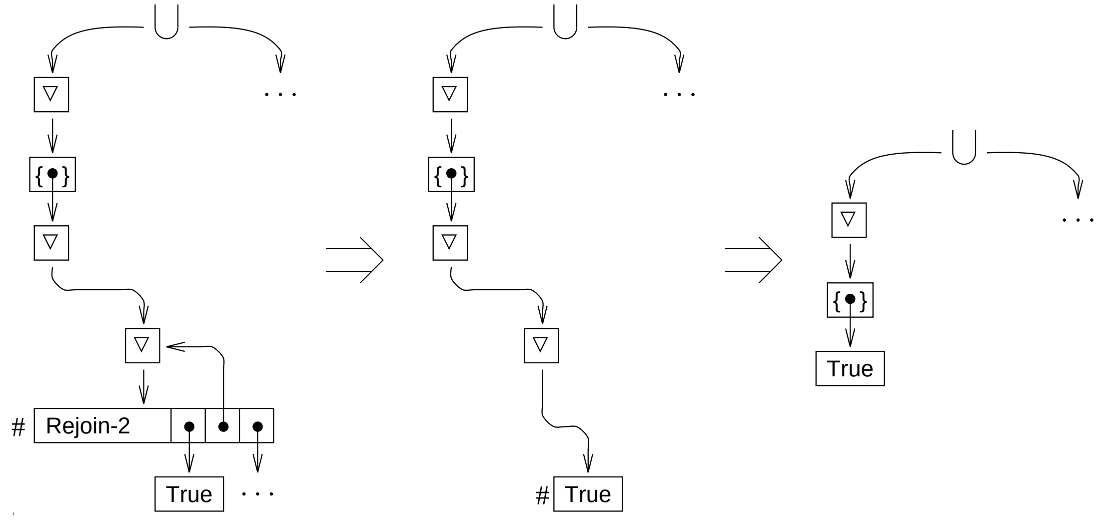


Figure 3.13: Downward propagation and removal of the Rejoin node from the graph of the running example under the naïve scheme for encountering Rejoin nodes.

is constructed, we cannot easily change *all* references to the original X within the bodies of the graphs of E_1 and E_2 into references to the new logic variables. These references are altered in the same way that Split-n and Rejoin-n nodes are moved through the graph, namely, only upon demand. If it becomes necessary to reduce either subexpression, nodes are checked for substitution as they are encountered. A new graph node, LVUpdate, will mark nodes which must be updated. Figure 3.14 shows the graph manipulation, which is easily seen as directly implementing this equivalent axiom:

$$set(X).(E_1 \cup E_2) = (set(Y).(E_1[Y/X])) \cup (set(Z).(E_2[Z/X]))$$

When a node tagged LVUpdate becomes the outermost redex, and its child node is *not* the logic variable pointed to by its first field, the LVUpdate node is propagated from above that child node to below it, above each of the child nodes' children, as described by Figure 3.15. Of course, if the child node *is* that logic variable, it is replaced by a reference to the new one and made the current redex, as in Figure 3.16.

3.3.3 Multiple Delimiters

The scheme outlined here does not hold quite enough information to work correctly in the not-unrealistic situation where narrowing has been performed on more than one logic variable. Propagating the Split node that results from a second (or subsequent) narrowing will be difficult: there will be Rejoin nodes corresponding to more than one logic variable, but there will be no way to determine, when encountering a Rejoin node during propagation, whether this Rejoin and Split node are associated with the same logic variable. This information is necessary since the propagation will proceed in two different ways based on this point (namely: it will either match the Rejoin and Split nodes, cease propagation and resume reduction, or otherwise duplicate the Rejoin node, and continue to propagate the Split node upwards).

A simple, elegant solution is to include some unique identifier in the Split and Rejoin nodes: if the identifiers are the same, the nodes correspond to the same logic variable. Any of several identifiers might be used:

1. Unique identifiers could be used, one generated at each occurance of narrowing.

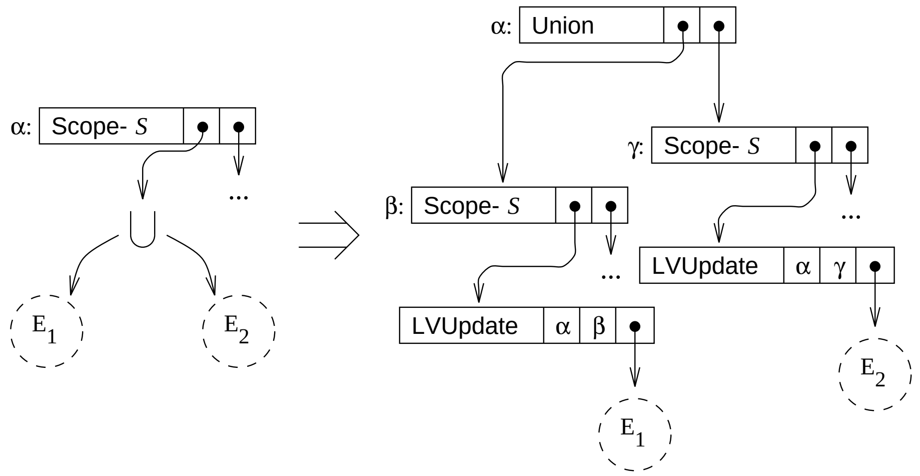


Figure 3.14: The graph reduction step for a union over a logic variable where the expression within the large union is itself a union of two sets.

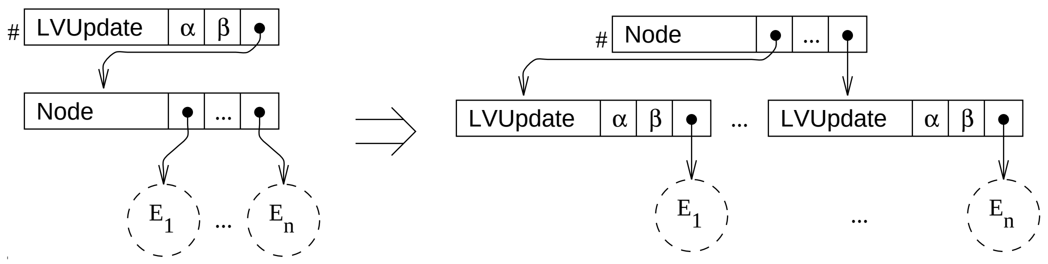


Figure 3.15: Distributing a LVUpdate node below the current redex a point where no updating needs to be made.

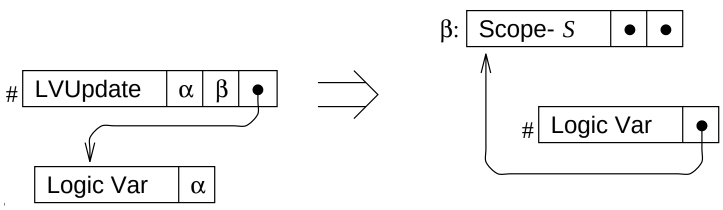


Figure 3.16: Performing an update required by an LVUpdate node.

2. The address of the original scope delimiter might be used.

3. The address of either the Split or Rejoin node created during narrowing might be used.

4. A pointer to some separate, unique graph node could be used.

For an abstract machine, without respect to actual implementation concerns, all of these alternatives are equally valid. But with an eye to an actual implementation, two other issues are relevant:

a. A separate generator for unique identifiers would require extra memory or registers, and extra time to use and maintain.

b. While the abstract machine has an arbitrarily large address space, an actual implementation will need to reuse addresses.

Point (a) means that option (1) will always require more machine overhead than the other options in an actual implementation. Point (b) means that if the identifier corresponds to a structure in the graph (as is true for the other options), that structure should not be reclaimed until no more references to Split nodes associated with that narrowing exist in the graph. Otherwise, if the address is reused, it could be used to store the graph node from which a subsequent narrowing takes *its* unique identifier, which would then mean that identifiers would no longer be unique. Thus option (2) is not viable, since the space occupied by a scope delimiter will be reclaimable after the initial propagation in narrowing (as per the description of the actual abstract machine, §4.5.4, p. 72 f.).

If no Split nodes corresponding to a certain narrowing are referenced in the graph after a certain point in the reduction, then obviously none will be found below the Rejoin nodes also associated with that same narrowing. With no quasisharing delimiters below them, the graphs under these Rejoin nodes are not quasishared, but are in fact fully shared. The conclusion of the earlier example is such a case: Figure 3.11 shows a Rejoin node with only a boolean-valued node below it; since no Split node is also below it the Rejoin node could be removed, with references to it treated as an indirection. Thus it could be argued that the address of the Split node should be used as the identifier, since it seems that when this node is no longer referenced, no more propagation will take place, and thus the Rejoin nodes are no longer needed: but this is not true, since the upward propagation of a Split node produces other Split nodes—associated with the same narrowing—which may need to be propagated themselves, and which may remain live in the graph longer than the original Split node. The observation above also lends itself to the argument that the address of the Rejoin node should *not* be used as the identifier, since it will be harder to determine when they should be discarded if Split nodes refer to Rejoin nodes, rather some form of the reverse.

Thus it would seem that option (4) is best: that the narrowing of a logic variable will produce, in addition to the structures already discussed, a graph node, tagged UID for unique identifier, to uniquely identify the quasisharing delimiters associated with that narrowing. Figure 3.17 updates Figure 3.3 to show this new structure: When a UID node is no longer referenced by any Split nodes due to those nodes' reclamation, the Rejoin nodes associated with it can also be reclaimed.

3.3.4 Self-Reference

It is possible that a well-defined set may be defined in terms of itself. In such a case, this scheme of scoping delimiters and logic variables must be handled carefully, since it may not be clear in such a case to which nested occurance of a scoping delimiter we should associate a logic variable. Consider the following example[4]:

```
letrec X be {cons(y,X): y∈hu}
    in {let a be car(A),
```

[4] Due to Silbermann, from a private communication

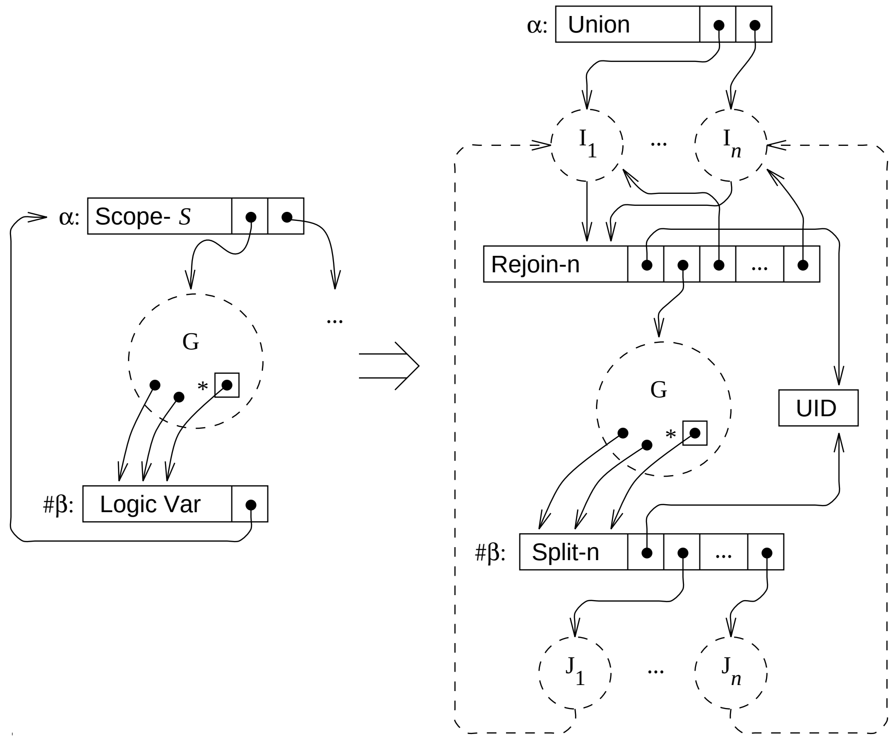


Figure 3.17: Introducing unique identifiers for narrowing structures when narrowing.

```
              b be car(B)
              in [a, b, a==b]: A∈X,
                               B∈cdr(A) }
```

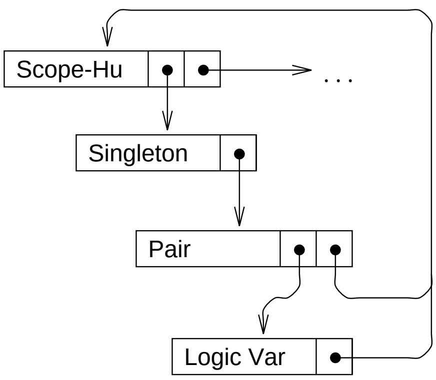
Lifted to supercombinators, we have:

```
$S = ⋃x∈hu {cons(x,$S)}
$B x y = let m = car(x)
             n = cdr(y)
           in {[m,n,m==n]}
$A x = ($B x)+cdr(x)
---
$A+$S
```

And then performing combinator reduction, we have the sequence, where the outermost redex of each term is underlined:

```
  $A+$S
⊢ $A+⋃x∈hu {cons(x,$S)}
⊢ ⋃x∈hu ($A+{cons(x,$S)})
⊢ ⋃x∈hu {$A cons(x,$S)}
⊢ ⋃x∈hu {($B cons(x,$S))+cdr(cons(x,$S))}
⊢ ⋃x∈hu {($B cons(x,$S))+$S}
⊢ ⋃x∈hu {($B cons(x,$S))+ (⋃x∈hu {cons(x,$S)})}
```

At this point in the combinator reduction, we would have to rename the bound quantifiers before contracting the indicated redex: failure to do so first would "capture" the `x` in the subterm `($B cons(x,$S))` — which clearly refers to the outer quantification — to make it refer to the inner quantification. If `X` were represented by the graph



then this mismatch of nested declaration and occurance is exactly what would result.

The easiest solution to this problem is to require that each reference in the graph to the supercombinator of an abstraction be required to construct its own version of the graph. The situation is analogeous to that for functions when they are represented explicity in the graph [7]. This task is easy to accomplish in the actual abstract machine when supercombinators are marked as described in §2.2.2; the implementation is decribed in §4.6 and §4.8.4.

3.4 Inequality Constraints

This section discusses the addition of inequality constraints on unbound quantities into the graph model for PowerFuL. Inequality constraints are necessary as a way to reduce to normal form certain boolean-valued expressions which involve unbound quantities. For example, in the expression

$$\bigcup_{x \in \mathbf{Atoms}} \textbf{if } x = \mathbf{A}_3 \textbf{ then } E_1 \textbf{ else } E_2$$

the outermost redex is $x = \mathbf{A}_3$. As is, this expression cannot be reduced into head normal form. To allow further reduction, we divide the expression into the union of two subexpressions which will allow further reduction. The basic rule to be used is:

$$(\bigcup_{x \in \mathbf{Atoms}} E) = \left(E\left[\frac{a}{x}\right]\right) \cup \left(\bigcup_{x \in \mathbf{Atoms},\, x \neq a} E\right)$$

which in this case gives the reduction:

$$\begin{aligned}
&(\bigcup_{x \in \mathbf{Atoms}} \textbf{if } x = \mathbf{A}_3 \textbf{ then } E_1 \textbf{ else } E_2) \\
&= \left(\textbf{if } \mathbf{A}_3 = \mathbf{A}_3 \textbf{ then } E_1\left[\frac{\mathbf{A}_3}{x}\right] \textbf{ else } E_2\left[\frac{\mathbf{A}_3}{x}\right]\right) \\
&\quad \cup \left(\bigcup_{x \in \mathbf{Atoms},\, x \neq \mathbf{A}_3} \textbf{if } x = \mathbf{A}_3 \textbf{ then } E_1 \textbf{ else } E_2\right) \\
&= \left(\textbf{if true then } E_1\left[\frac{\mathbf{A}_3}{x}\right] \textbf{ else } E_2\left[\frac{\mathbf{A}_3}{x}\right]\right) \\
&\quad \cup \left(\bigcup_{x \in \mathbf{Atoms},\, x \neq \mathbf{A}_3} \textbf{if false then } E_1 \textbf{ else } E_2\right) \\
&= \left(E_1\left[\frac{\mathbf{A}_3}{x}\right]\right) \cup \left(\bigcup_{x \in \mathbf{Atoms},\, x \neq \mathbf{A}_3} E_2\right)
\end{aligned}$$

So that, assuming E_1 and E_2 are well-defined, reduction to normal form may continue.

Representation

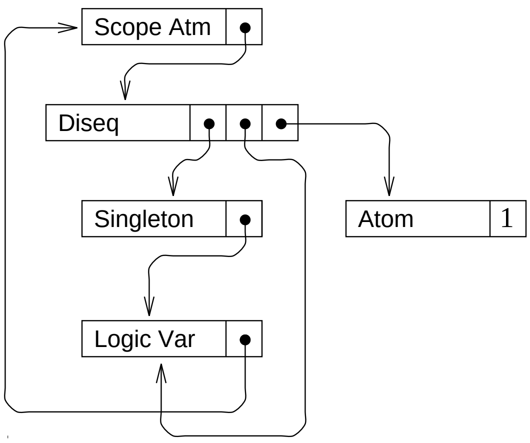
In the optimized denotational semantics of PowerFuL [10], inequality contraints are expressed in the following manner:

$$set(X).X \neq x_1.X \neq x_2.\ldots.X \neq x_n.E$$

This notation directly suggests a representation for inequalities in a graph model, where inequalities may be stored as subexpressions of their declaration. For example, the expression

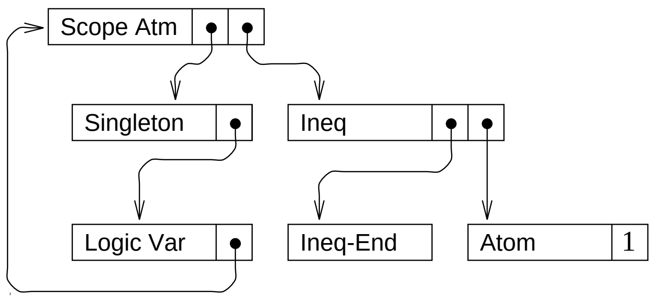
$$\bigcup_{x\, \in \mathbf{atoms},\; x\, \neq \mathbf{A}_1} \{x\}$$

would be modeled as



This scheme has one important disadvantage: traversing the graph to find the outermost redex under the first scheme would require passing through *each* inequality node associated with the logic variable, such a traversal might have to be done several times.

As an improvment to this scheme, inequalities might be stored with the declaration of the logic variable, *not* as a subexpression, but rather simply in a list of their own. Under such a scheme the above example would be represented by



Unfortunately, this scheme is not sufficient for general use. Consider this example:

$$\bigcup_{x\in\mathbf{hu}} \bigcup_{y\in\mathbf{hu},\ \mathbf{y\neq x}} \textbf{if } y\in b \textbf{ then } \{\mathbf{x}\} \textbf{ else } \phi$$

The outermost redex here is $y \in b$, and therefore narrowing is necessary. The relevant narrowing rule is

$$\bigcup_{x\in\mathbf{hu}} E \Longrightarrow \bigcup_{y\in\mathbf{atoms}} E\left[\frac{y_a}{x}\right] \cup \left(\bigcup_{y\in\mathbf{bools}} E\left[\frac{y_b}{x}\right]\right) \cup \left(\bigcup_{y_1\in\mathbf{hu}}\bigcup_{y_2\in\mathbf{hu}} E\left[\frac{\langle y_1,y_2\rangle_p}{x}\right]\right)$$

But inequality relationships must also be preserved: by Silbermann and Jayaraman [10], the inequality would be part of E, a more appropriate statement of the rule would be

$$\begin{aligned}\bigcup_{x\in\mathbf{hu},\, x\neq A} E \Longrightarrow\ & \left(\bigcup\nolimits_{y\in\mathbf{atoms},\, y_a\neq\left(A\left[\frac{y_a}{x}\right]\right)} E\left[\tfrac{y_a}{x}\right]\right)\\ & \cup \left(\bigcup\nolimits_{y\in\mathbf{bools},\, y_b\neq\left(A\left[\frac{y_b}{x}\right]\right)} E\left[\tfrac{y_b}{x}\right]\right)\\ & \cup \left(\bigcup\nolimits_{y_1\in\mathbf{hu}} \bigcup\nolimits_{y_2\in\mathbf{hu},\, \langle y_1,y_2\rangle_p\neq\left(A\left[\frac{\langle y_1,y_2\rangle_p}{x}\right]\right)} E\left[\tfrac{\langle y_1,y_2\rangle_p}{x}\right]\right)\end{aligned}$$

which in this case gives

$$\begin{aligned}& \bigcup_{y\in\mathbf{hu},\, y\neq x} \textbf{if } y\in b \textbf{ then } \{\mathbf{x}\} \textbf{ else } \phi\\ \Longrightarrow\ & \left(\bigcup_{z\in\mathbf{atoms},\, z_a\neq x} \textbf{if } z_a\in b \textbf{ then } \{\mathbf{x}\} \textbf{ else } \phi\right)\\ & \cup \left(\bigcup_{z\in\mathbf{bools},\, z_b\neq x} \textbf{if } z_b\in b \textbf{ then } \{\mathbf{x}\} \textbf{ else } \phi\right)\\ & \cup \left(\bigcup_{z_1\in\mathbf{hu}} \bigcup_{z_2\in\mathbf{hu},\, \langle z_1,z_2\rangle_p\neq x} \textbf{if } \langle z_1,z_2\rangle_p\in b \textbf{ then } \{\mathbf{x}\} \textbf{ else } \phi\right)\end{aligned}$$

The situation of the inequalities at this point is awkward at best: the intended model of the logic variable was that each variable's definition would contain a list of quantities to which that logic variable would never be equal. However, the result of narrowing is that instead of the defined logic variable itself not being equal to the list's elements, it is an expression over the defined variable to which the elements in the list apply. This situation may arise anytime a non-ground quantity appears in an inequality list, so it will be necessary to restrict inequality lists to simple constant expressions. In fact since

- we need never record inequalities involving quantifiers ranging over **bools**, in which case it is just as efficient to narrow into the two constituant cases, and
- variables over **hu** are only ever declared to be not equal to other logic variables over **hu**

it is possible to restrict membership in inequality lists to only Atom nodes! This restriction will make their implementation more conservative of space, since the Atom tag can now be eliminated entirely, and only the atom's number stored. Since these lists are known to be constant, no substitution ever need be done across them.

Thus, the best model for inequality constraints is a combination of the two possible models. Whenever possible, the more efficient lists can be used, but constraints involving more than one logic variable will be stored as subexpressions.

As is the case with other nodes, inequality nodes can duplicated by Split and Rejoin nodes. Lists can be shared among split scoping declarations so long as subsequent additions to individual nodes' lists are prepended, and simply refer to the shared list, rather than being added afterwards, altering the original.

Chapter 4

NG: An Abstract Machine for PowerFuL

Having presented a graph model for implementing PowerFuL, it is appropriate to consider the sort of abstract machine which should be used to implement the model. The graph model relies on the ability to access and manipulate the spines of application nodes which make up expressions. Thus, it is essential that whatever abstract machine is used build and maintain application spines. Of the spine-based abstract machines used for implementing functional languages, the Chalmers G-machine is among the more elegant and straightforward designs. The basic G-machine implements graph reduction almost literally, but still quite efficiently.

This chapter presents an abstract machine for compiling and executing PowerFuL programs based on the G-machine. The new machine is called the NG-machine, short for the Narrowing G-machine. The new machine is developed from the old in several steps: first, basic set constructs are given a graph representation and NG-code instructions for their construction and analysis. A new notion of weakened normal form is then defined, to describe the needed manipulations to graphs of expressions denoting sets, and the evaluation mechanism adjusted to reduce to that normal form. Finally, the machine is extended to narrow expressions when appropriate.

This chapter describes the derivation of the NG-machine from the G-machine of §2.3. For standard G-machine commands, the derivation rules for the NG-machine are taken to be the same as those for the G-machine except where specified otherwise.

4.1 Code References on the Dump

Narrowing will sometimes require that previously executed commands be re-executed in a modified machine configuration (§4.4.2, §4.5.4). To facilitate this effect, a slightly different protocol for storing code sequences on the dump is appropriate. Rather than removing the first instruction from a sequence before storing it on the dump, the sequence will be stored with that instruction, and subsequent removal of the sequence from the dump will also strip the leading instruction from the sequence.

To illustrate the difference, consider the evaluation of an application node in the standard G-machine. The current code and stack pointers are stored on the dump while the application spine is unwound, as per the derivations given in §2.3.2:

$$(\texttt{Eval}; C,\, a{:}S,\, G,\, D) \vdash (\texttt{Unwind},\, a{:}[\,], G,\, \langle S, C\rangle{:}D) \vdash \ldots \vdash (C,\, a{:}S,\, G,\, D)$$

Under the new calling convention, the NG-machine would follow the same sequence with these configurations:

$$(\texttt{Eval}; C,\, a{:}S,\, G,\, D) \vdash (\texttt{Unwind},\, a{:}[\,], G,\, \langle S, \texttt{Eval}; C\rangle{:}D) \vdash \ldots \vdash (C,\, a{:}S,\, G,\, D)$$

The difference is only in when the "instruction register" is increased. The use of the new convention will facilitate repeating instructions:

$$\begin{aligned}(\texttt{Eval}; C,\ a{:}S,\ G,\ D) &\vdash (\texttt{Unwind},\ a{:}[\,],\ G,\ \langle S, \texttt{Eval}; C\rangle{:}D) \vdash \ldots \\ &\vdash (\ldots,\ a{:}[\,],\ G,\ \langle S, \texttt{Eval}; C\rangle{:}D) \vdash (\texttt{Eval}; C,\ a{:}S,\ G,\ D)\end{aligned}$$

The usefulness of re-executing a command will become apparent in the subsequent discussions.

Two cases of the `Eval` command alter the dump and must be redefined,

$$(\texttt{Eval}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ a_1\ a_2 \vdash (\texttt{Unwind},\ n{:}[\,],\ G,\ \langle S, \texttt{Eval}; C\rangle{:}D)$$
$$(\texttt{Eval}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Fun}\ 0\ C' \vdash (C',\ n{:}[\,],\ G,\ \langle S, \texttt{Eval}; C\rangle{:}D)$$

while the other two are unchanged,

$$(\texttt{Eval}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Eval}; C,\ a'{:}S,\ G,\ D)$$
$$(\texttt{Eval}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} N \vdash (C,\ a{:}S,\ G,\ D),$$

where

$$N \in \{\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \mathsf{Fun}\ n\ i,\ n > 0\ \}$$

(N.B., these rules for `Eval` *will be revised further in §4.4, and are* not *those used in the final NG-machine.)* The `Unwind` command, which also alters the dump, is redefined as follows:

$$(\texttt{Unwind}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ n'\ m \vdash (\texttt{Unwind}; C,\ n'{:}n{:}S,\ G,\ D)$$
$$\begin{aligned}(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_m{:}S,\ G,\ D),\ a_0 \xrightarrow{G} \mathsf{Fun}\ m\ C,\ a_i &\xrightarrow{G} \mathsf{Ap}\ a_{i-1}\ n_i,\ 1 \le i \le m \\ &\vdash (C,\ n_1{:}\ldots{:}n_m{:}a_m,\ G,\ \langle S, \texttt{Unwind}; \texttt{Unwind}\rangle{:}D)\end{aligned}$$
$$\begin{aligned}(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_k,\ G,\ \langle S, x; C\rangle{:}D),\ a_0 &\xrightarrow{G} \mathsf{Fun}\ m\ C_1,\ k < m \\ &\vdash (C,\ a_k{:}S,\ G,\ D)\end{aligned}$$
$$(\texttt{Unwind}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Unwind}; C,\ a'{:}S,\ G,\ D)$$
$$(\texttt{Unwind}; C_0,\ n{:}[\,],\ G,\ \langle S, x; C\rangle{:}D),\ n \xrightarrow{G} N \vdash (C,\ n{:}S,\ G,\ D),$$

where

$$N \in \{\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \}.$$

The `Return` command becomes:

$$(\texttt{Return}; C_0,\ [\,],\ G,\ \langle S, x; C\rangle{:}D) \vdash (C,\ S,\ G,\ D)$$
$$(\texttt{Return}; C_0,\ a{:}[\,],\ G,\ \langle S, x; C\rangle{:}D) \vdash (C,\ a{:}S,\ G,\ D)$$

Finally, the printing mechanism must be modified as follows:

$$(O,\ \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Atom}\ \alpha \vdash (O; \alpha,\ C,\ S,\ G,\ D)$$
$$(O,\ \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Bool}\ \beta \vdash (O; \beta,\ C,\ S,\ G,\ D)$$
$$(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a' \vdash (\texttt{\_Print\_Tagged\_}\tau,\ a'{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}D)$$
$$(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 \vdash (\texttt{\_Print\_Pair},\ a_1{:}a_2{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}D)$$
$$(O,\ \texttt{PrintTag}\ \ \tau; C,\ S,\ G,\ D) \vdash (O; \tau,\ C,\ S,\ G,\ D).$$

4.2 Basic Set Manipulation

The facility to manipulate basic set constructs is added to the G-machine with new graph node tags and G-code commands in the spirit of the standard tags and commands. On the surface, the new tags and commands seem simply to form no more than a new data type, which could be equally well modeled by pairs and distinguished atoms. However, the interaction of the new constructs with the standard commands which an implementation faithful to the optimized denotational semantics must allow distinguishes the new constructions from simple data groupings: manipulations such as narrowing, and the intricacies of printing set-denoting expressions, will be facilitated by their distinct representation.

4.2.1 Set Constructs and Derivations

The graph node types outlined in §3.1 (p. 25) can be converted directly into G-machine graph node types. The empty set is modeled by a childless node labeled Phi. As with atoms and boolean values, the empty set can be simply pushed onto the stack:

$$(\texttt{Push Phi}; C,\, S,\, G,\, D) \vdash (C,\, a{:}S,\, G[a \mapsto \textsf{Phi}],\, D)$$

For singleton sets, a graph representation is made by a one-child node tagged Singleton, plus two new G-code instructions to form a set from a value and to return the value contained in a set. The former command is `MkSingleton`, defined by

$$(\texttt{MkSingleton}; C,\, a{:}S,\, G,\, D) \vdash (C,\, a'{:}S,\, G[a' \mapsto \textsf{Singleton}\ a],\, D)$$

The latter command is `SingletonElement`, defined by

$$(\texttt{SingletonElement}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Singleton}\ a' \vdash (C,\, a'{:}S,\, G,\, D)$$

Unions of sets are modeled by a two-child node tagged Union. Unions are built with the `MkUnion` command, defined by

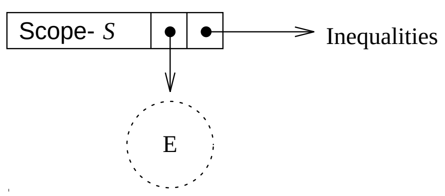
$$(\texttt{MkUnion}; C,\, a_1{:}a_2{:}S,\, G,\, D) \vdash (C,\, a'{:}S,\, G[a' \mapsto \textsf{Union}\ a_1\ a_2],\, D),$$

and taken apart with the `UnionElements` command,

$$(\texttt{UnionElements}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Union}\ a_1\ a_2 \vdash (C,\, a_1{:}a_2{:}S,\, G,\, D)$$

4.2.2 Logic Variables

Logic variables are modeled in the extended G-machine as described in §3.2. The graph node



denotes the top of the scope of a logic variable ranging over the set S; there are thus three different such node types:

1. the set **hu**, the Herbrand Universe, is modeled with a node tagged Scope Hu;
2. the set of atomic values **atoms** uses a node tagged Scope Atm;
3. the set of boolean values **bools** uses a node tagged Scope Bool.

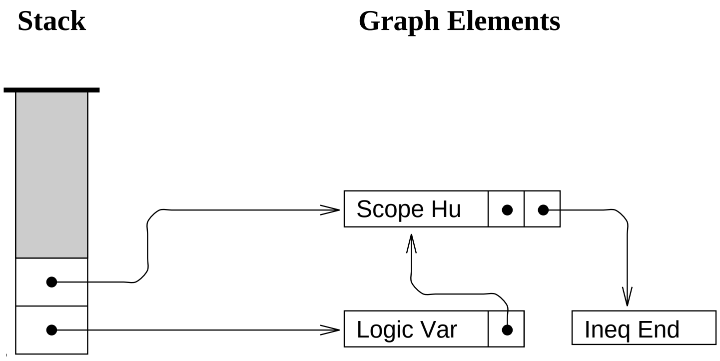


Figure 4.1: Stack configuration after the `NewLVar` command.

Associated with each occurance in the graph of a node of these types is a node labeled Logic Var, which is the unique graph node representing the value declared in that scope. Logic variables and their scope delimiters are introduced by the `NewLVar` command,

$$(\texttt{NewLVar}; C, S, G, D) \vdash (C, a_1{:}\, a_2{:}\, S, G \begin{bmatrix} a_1 \mapsto \mathsf{Logic\ Var}\ a_2 \\ a_2 \mapsto \mathsf{Scope{-}Hu}\ \perp\ a_3 \\ a_3 \mapsto \mathsf{Ineqs{-}End} \end{bmatrix}, D)$$

To complete such a definition, the `CompleteLVScope` command stores the address of a graph node in the first slot of a scope delimiter.

$$\begin{aligned} &(\texttt{CompleteLVScope}; C, a_1{:}\, a_2{:}\, S, G, D),\ a_2 \xrightarrow{G} \mathsf{Scope{-}Hu}\ m\ n \\ \vdash\ \ &(C, a_2{:}\, S, G[a_2 \mapsto \mathsf{Scope{-}Hu}\ a_1\ n], D) \end{aligned}$$

While this scheme for introducing logic variables is somewhat convoluted, it does facilitate the construction of expressions over logic variables, which are usually circular graphs since a logic variable will refer back up to the node which declares its scope. The execution of the `NewLVar` command will leave the stack in the configuration shown in Figure 4.1. In constructing the nested expression, subsequent G-code would be expected to leave the stack as shown in Figure 4.2, where the expression represented by E would likely contain one or several references to the logic variable. Finally, the `CompleteLVScope` command would move from that configuration to the one shown in Figure 4.3.

The subexpression over which a scoped declaration ranges is accessed with the `GetScopedValue` command:

$$(\texttt{GetScopedValue}; C, a{:}\, S, G, D),\ a \xrightarrow{G} \mathsf{Scope{-}}S\ a'\ I \vdash (C, a'{:}\, S, G, D)$$

4.2.3 Inequality

The implementation of the graph model for inequalities of §3.4 requires three new types of graph node. A Diseq node stores pairs of values which must not be equal to each other; Diseq is used as a subexpression of a Scope node and is thus a set-denoting node. Ineq and IneqEnd are used as, respectively, the links and the terminator in a list of inequalities associated with the declaration of a logic variable ranging over atoms. Thus, the expression

$$\bigcup_{x \in \mathbf{atoms},\, x \neq \mathbf{A}_4} E$$

can be expressed as the graph shown in Figure 4.4, while the expression

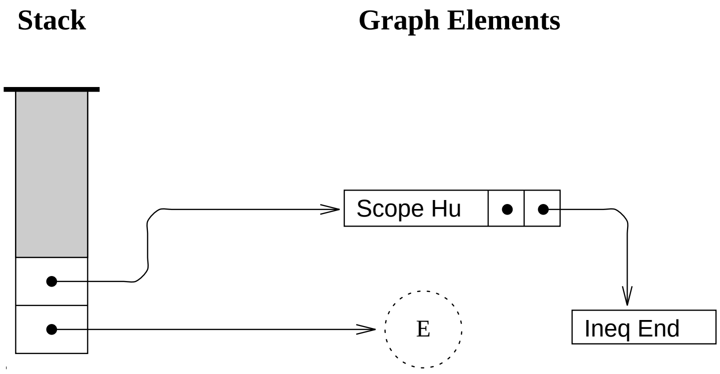


Figure 4.2: Stack set up for the `CompleteLVScope` command.

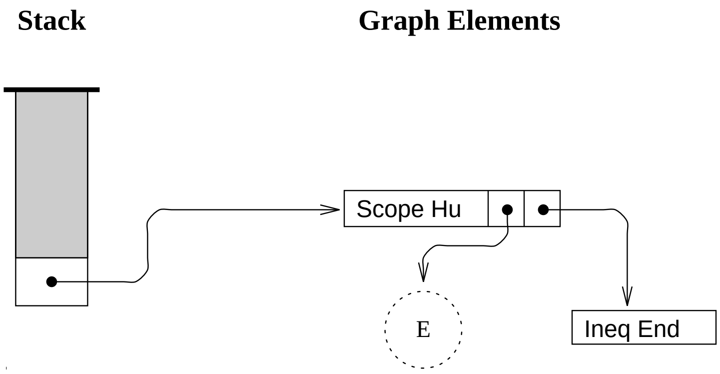


Figure 4.3: Stack configuration after the `CompleteLVScope` command.

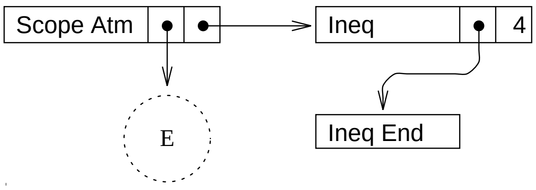


Figure 4.4: Graph representation of inequality with simple, constant values.

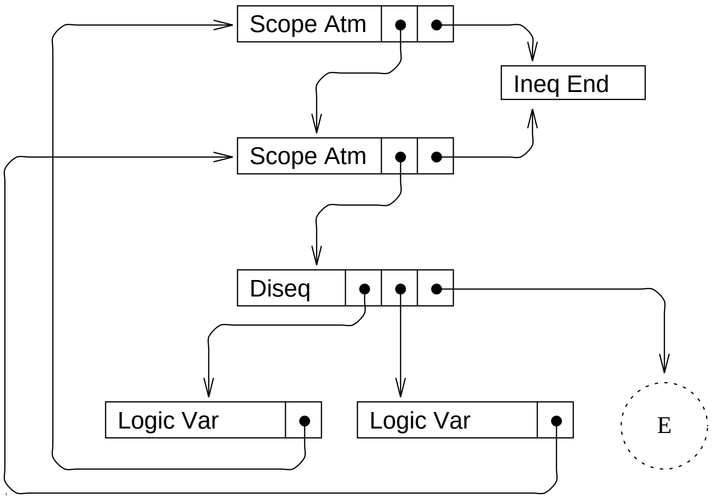


Figure 4.5: Graph representation of inequality between logic variables.

$$\bigcup_{x \in \mathbf{atoms}} \bigcup_{y \in \mathbf{atoms}} x \neq y.E$$

can be represented as the graph shown in Figure 4.5. The subexpression over which a Diseq declaration ranges is accessed with the `GetDisequaledSubexpr` command,

$$(\texttt{GetDisequaledSubexpr}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a' \vdash (C,\, a'{:}S,\, G,\, D),$$

the two expressions identified as unequal can be accessed using the `GetTwoDisequals` command,

$$(\texttt{GetTwoDisequals}; C,\, a{:}S,\, G,\, D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a'\ \vdash\ (C,\, a_1{:}a_2{:}S,\, G,\, D)$$

and the nodes constructed with the `MkDisequal` command:

$$(\texttt{MkDisequal}; C,\, a_1{:}a_2{:}a'{:}S,\, G,\, D)\ \vdash\ (C,\, a{:}S,\, G[a \mapsto \mathsf{Diseq}\ a_1\ a_2\ a'],\, D).$$

4.2.4 Distinguishing Set Nodes

This representation of set values is more complicated than the representation for other values in the sense that the representation of a well-formed expression with embedding marker a, b or p, when reduced to weak-head normal form, will be a node whose tag is uniquely identified with the particular embedding marker. That is, the node for an expression under a $\mathsf{Embed}{-}a$ node will have node tag Atom when evaluated; under $\mathsf{Embed}{-}b$, Bool; under $\mathsf{Embed}{-}p$, Pair. Some of the transactions we will perform on graphs representing set-valued expressions will need to distinguish between the three types of set-representing nodes. The `SetCase` command is used to distinguish between the different set-denoting nodes and tags by using a four-way branch:

$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ \ a \xrightarrow{G} \mathsf{Phi}$$
$$\vdash\ (L_1,\, a{:}S,\, G,\, D)$$
$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ \ a \xrightarrow{G} \mathsf{Singleton}\ a'$$
$$\vdash\ (L_2,\, a{:}S,\, G,\, D)$$

$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \textsf{Union}\ a_1\ a_2$$
$$\vdash\ (L_3,\ a{:}S,\ G,\ D)$$
$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \textsf{Scope}{-}S\ a'\ I$$
$$\vdash\ (L_4,\ a{:}S,\ G,\ D)$$
$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \textsf{Diseq}\ e_1\ e_2\ a'$$
$$\vdash\ (L_4,\ a{:}S,\ G,\ D)$$
$$(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C,\ a'{:}S,\ G,\ D)$$

4.3 Notions of Normal Form

In the graph reduction of pure functional languages, the aspect of lazy evaluation dealing with normal order evaluation is implemented in part with the concept of weak head normal form (WHNF) [7]. The concept of WHNF captures, in terms of the structure of an expression, the property that an expression is not itself a redex, although it may contain proper subexpressions which are redexes. For the graph representation of an expression, being in WHNF means that the uppermost node of the graph is either

1. a data constructor node, or
2. a function of non-zero arity.

For values in functional languages, an expression in WHNF also has the property that the constructor node at the top of the graph will not change: a list will always be a list, a function node of non-zero arity is the final form of such an expression, and so on.

The practical value of this concept is that the printing mechanism can take advantage of the fact that these top nodes are fixed by printing them and disposing of them, evaluating and printing subexpressions in turn. Using WHNF in this way allows for a very elegant `Eval-Print` structure to the abstract machine: the former command produces exactly what the latter needs to work, and no more.

Unfortunately, for set-denoting expressions this notion of normal form does not coincide with the certainty that the outermost tag of the graph of an expression is fixed. For example, while union nodes are often not reduced to another node, such as in the expression

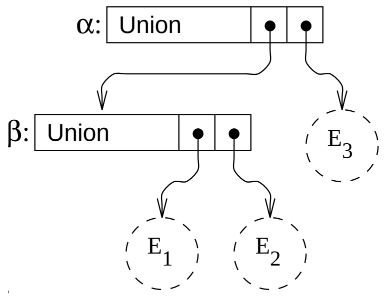
$$\{\mathbf{a}\} \cup \{\mathbf{b}\},$$

a union node could be reduced to a singleton set ($\{\mathbf{b}\} \cup \phi \Longrightarrow \{\mathbf{b}\}$) or to the empty set ($\phi \cup \phi \Longrightarrow \phi$).

There are two sensible ways to overcome this difficulty:

1. Develop two distinct concepts of union in the graph. The union symbol in the syntax/semantics will translate into the application of a *union function*, which may reduce to a *union node*.
2. Identify a different notion of weak normal form which accurately captures the required behavior to facilitate printing.

Upon further inspection, it turns out that the first option is not viable. First, such a separation leaves unsolved a problem with the printing mechanism: A union of sets should displayed to a program's user in a specific manner: since a tree of union nodes may be evaluable to arbitrarily many singleton sets, it is necessary to give the user the opportunity to halt execution after the generation of any individual singleton set. This effect can be produced by, when printing a union node, printing the left subexpression, querying the user as to whether additional set elements should be printed, and if so, printing the right subexpression. A consequence of this scheme is that the left child of an evaluated Union node should *never* be a union itself. Consider the printing of the node α:



Printing the node α will immediately cause the printing of the node β. If the results of the user query in this second run, on β, were negative, then *all* evaluation of the sets in the union should stop, but it will not necessarily do so: the first request to print α would then resume, and issue a second query regarding further generation of the same overall union. This effect could be a mere inconvenience, but in the case of nested sets, it could be a point of significant confusion.

The problem could be resolved with a clever definition of the union function, but problems still remain. The first problem is the consumption of heap space in building application nodes which will almost immediately be reduced: but since partial applications may be shared, these nodes may not be reclaimable. Secondly, interaction with the scoping delimiters as subexpressions of a union node or as arguments to the union function will again cause complications, since subsequent reduction of these nodes may generate a union of sets, reintroducing the multiple prompt problem. Again, a clever definition of the union function could alleviate this situation, but at the cost of greatly increased complexity. The last objection is that expressions like

$$(\{\mathbf{a}\} \cup \{\mathbf{b}\}) \cup (\{\mathbf{c}\} \cup \{\mathbf{d}\})$$

are in normal form, and thus an extension to the G-machine should be able to handle them. An extension which requires a contrived translation scheme to avoid generating such expressions would be both inelegant and weak, and could be limiting to future modifications. Opting for the second alternative, and devising a new notion of weakened normal form, gives rise to an elegant implementation (§4.4) in addition to satisfying all of these objections.

Print-Ready Weak Normal Form

The first form considered is as follows:

Definition 4.1 *An expression is said to be in* union-excepted print-ready weak normal form *(PRWNF$_\cup$) if and only if it is of the form*

$$E_0\ E_1\ \ldots\ E_m$$

where the following conditions are all true:

- $m \geq 0$
- *either*
  - E_0 *is a variable or data object, or*
  - E_0 *is a* λ*-abstraction or function of arity* n*, with* $m < n$
- *no reduction of any* E_i *will produce a narrowing which alters or duplicates* E_0

The definition of WHNF, for the sake of comparison, is the same except that it omits the last requirement. This last item is necessary for printing expressions ranging over logic variables. If the machine considered that given an expression

$$\bigcup_{x \in \mathbf{hu}} E,$$

for *any, arbitrary* E, it is appropriate to generate to the output

`U_(x∈hu)`,

a problem would arise if E contained an occurrance of x which required narrowing of its scope. For example, the expression

$$\bigcup_{x \in \mathbf{hu}} \langle x \in a, x \rangle_p$$

is in weak head normal form, but printing `U_(x∈hu)` upon seeing this expression would be inappropriate, since when reduced to full normal form this expression becomes

$$\bigcup_{y \in \mathbf{atoms}} \langle \mathbf{true}, x_a \rangle_p \cup \bigcup_{y \in \mathbf{bools}} \langle \mathbf{false}, x_b \rangle_p \cup \bigcup_{y \in \mathbf{hu}} \bigcup_{z \in \mathbf{hu}} \left\langle \mathbf{false}, \langle y, z \rangle_p \right\rangle_p,$$

which would require completely different output.

The definition above satisfies the need to make sure that the outermost node will not be altered, but it does not take care of the needed restrictions on union nodes. The following form meets this additional criterion:

Definition 4.2 *An expression is in* print-ready weak normal form *(PRWNF) if and only if*

- *it is in PRWNF*$_\cup$*, and*
- *it is of the form*

$$E_0 \; E_1 \; \ldots \; E_n$$

where the following two conditions are not both true:

- E_0 *denotes, or will reduce to an expression which denotes, a union operator, and*
- E_1 *denotes, or will reduce to an expression which denotes, a binary union of sets.*

The last clause of this definition precludes the left subgraph of a union node from being a binary union node, and thus alleviates the problem of multiple queries for generating union elements.

4.4 Standard Reduction Sequences

Admittedly, the normal forms above are somewhat contrived. However, they do provide a concise, formal expression of how the new abstract machine should behave. The main action of the machine should be to reduce an expression to PRWNF (instead of WHNF) and then print it. Rather than redefining existing evaluation commands, a much more elegant implementation is possible using multistage evaluation: where the standard G-machine uses a single command for evaluation, the NG-machine will use different commands to reduce expressions to different normal forms. Akin to the `Eval` command in the standard G-machine, the `Eval-wh` command reduces graph expressions to WHNF. In addition, the `Eval-prw` command reduces expressions already in WHNF to PRWNF, and the `Eval-f` command reduces expressions already in WHNF to full normal form (FNF).

4.4.1 Reduction to Weak Head Normal Form

The definition of `Eval` is trivially extended to the new tags in `Eval-wh`; since all of the new tags could be present in final graph forms, their presence indicates an expression in WHNF:

$$(\texttt{Eval-wh}; C,\, n{:}S,\, G,\, D),\; n \xrightarrow{G} N \vdash (C,\, n{:}S,\, G,\, D),$$

where

$$N \in \{ \begin{array}{l} \textsf{Phi}, \\ \textsf{Singleton}\ a, \\ \textsf{Union}\ a_1\ a_2 \\ \textsf{Logic Var}\ a, \\ \textsf{Scope}{-}S\ a\ I, \\ \textsf{Diseq}\ e_1\ e_2\ a, \\ \textsf{Atom}\ \alpha, \\ \textsf{Bool}\ \beta, \\ \textsf{Pair}\ a_1\ a_2 \\ \textsf{Embed}{-}\tau\ a \\ \textsf{Fun}\ n\ i,\ n > 0 \quad \} \end{array}$$

The remaining rules for `Eval-wh` are simply the rules for `Eval` transliterated to refer to the new command:

$$(\texttt{Eval-wh}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \textsf{Ap}\ a_1\ a_2 \ \vdash\ (\texttt{Unwind},\ n{:}[\,],\ G,\ \langle S, \texttt{Eval-wh}; C\rangle{:}D)$$

$$(\texttt{Eval-wh}; C_0,\ n{:}S_0,\ G,\ D),\ n \xrightarrow{G} \textsf{Fun}\ 0\ C \ \vdash\ (C,\ n{:}[\,],\ G,\ \langle S_0, \texttt{Eval-wh}; C_0\rangle{:}D)$$

$$(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \ \vdash\ (\texttt{Eval-wh}; C,\ a'{:}S,\ G,\ D)$$

4.4.2 Reduction to Print-Ready Weak Normal Form

For reduction to PRWNF, the `Eval-prw` command is used *following* the application of the `Eval-wh` command: consistant with the idea of multistage evaluation, `Eval-prw` performs a case analysis on the tag of the node pointed to by the top of the stack:

- The `Eval-prw` command will do nothing, and simply proceed to the next command, in the case of
  - Node tags which denote a singleton set: singleton sets will only be narrowed in the context of a scoped expression delimiter, but never on their own.
  - Non-set values, which are ready to print out when they are in WHNF.
  - The empty set, which as a constant value is also ready to print.
  - A logic variable, which is taken to be in full normal form.

  The rule is then:

  $$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} N \vdash (C,\ a{:}S,\ G,\ D)$$

  where

  $$N \in \{ \begin{array}{l} \textsf{Phi}, \\ \textsf{Singleton}\ a, \\ \textsf{Logic Var}\ a, \\ \textsf{Atom}\ \alpha, \\ \textsf{Bool}\ \beta, \\ \textsf{Pair}\ a_1\ a_2 \\ \textsf{Embed}{-}\tau\ a \\ \textsf{Fun}\ n\ i,\ n > 0 \quad \} \end{array}$$

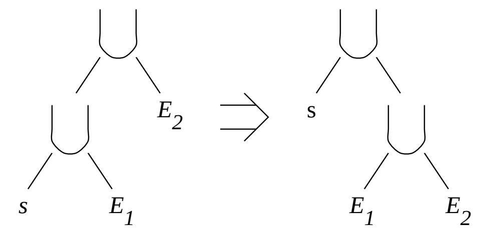
- Indirection nodes, as usual, must be dereferenced:

  $$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Eval-prw}; C,\ a'{:}S,\ G,\ D)$$

```
_E2_Union:  Push 0
            UnionElements
            Eval-wh
            Eval-prw
            SetCase _E2_Be _E2_Bss _E2_Bun _E2_Bss
   _E2_Be:  Pop 1
            Update 1
            Return
  _E2_Bss:  Pop 2
            Return
  _E2_Bun:  UnionElements
            Push 2
            Push 2
            MkUnion
            Push 1
            MkUnion
            Update 4
            Pop 3
            Return
```

Figure 4.6: The code sequence `_E2_Union`.

- Union nodes must be manipulated to make sure that exactly *one* set expression will be printed, and neither a union of more that one set nor the empty set. This effect is achieved by, in an auxiliary code sequence, reducing the left subexpression to PRWNF and performing an analysis of the resulting node's tag. If it is the empty set, the union can be reduced to its right subexpression; if a singleton set or a scoped expression delimiter, then the entire expression is in PRWNF; or if another union, then the group of three items being united must be rearranged so that the left child of the lower node be made the left child of an upper node:

  The rules for this case are:

  $$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Union}\ a_1\ a_2 \vdash (\texttt{\_E2\_Union},\ a{:}[\,],\ G,\ \langle S, C\rangle{:}D)$$

  where `_E2_Union` is the code sequence shown in Figure 4.6.

- Reducing the delimiter of a scoped expression to PRWNF involves several steps. If the scoped expression turns out to be a binary union, then the axiom

  $$\bigcup(X \cup Y) = \bigcup X \cup \bigcup Y$$

  should be applied; if the scoped expression is the empty set, then the axiom

```
_force_scoped:  Push 0
                GetScopedValue
                Eval-wh
                SetCase _fs_e _fs_f _fs_bu _fs_f
        _fs_e:  Update 1
                Return
       _fs_bu:  UnionElements
                NewLVar
                Push 4
                Push 4
                MkLVarSwap
                CompleteLVScope
                NewLVar
                Push 5
                Push 4
                MkLVarSwap
                CompleteLVScope
                MkUnion
                Update 3
                Pop 2
                Retry
       _fs_bu:  Eval-f
                SetCase _fs_e _fs_ok _fs_bu _fs_ok
       _fs_ok:  Pop 1
                Return
```

Figure 4.7: The code sequence `_force_scoped`.

$$\bigcup \phi = \phi$$

applies. Otherwise, the scoped expression should be reduced to full normal form, and the same tests repeated. The goal of reduction to PRWNF for a scoped expression's delimiter is to make sure that it will not be updated with a union node. To do this, one must be able to determine that no redex involving a logic variable will ever require more information than is inherant in the logic variable itself. Since a traversal of the entire subgraph would be very expensive, it is necessary to reduce the entire subexpression to full normal form: this is the effect of `Eval-f`, so we can use that command in an auxiliary code sequence:

$$\begin{array}{l}(\texttt{Eval-prw}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Scope}{-}S\ a'\ I \\ \quad \vdash\ (\texttt{\_force\_scoped},\, a'\!:\![\,],\, G,\, \langle a\!:\!S, \texttt{Eval-prw}; C\rangle\!:\!D)\end{array}$$

Figure 4.7 gives the code sequence `_force_scoped`. The `Retry` command used in the sequence exploits the calling configuration of §4.1:

$$(\texttt{Retry}; C_0,\, S_0,\, G,\, \langle S, C\rangle\!:\!D)\ \vdash\ (C,\, S,\, G,\, D)$$

This reasoning holds in part for disequality nodes, whose treatment we defer to §4.5.4, p. 78.

4.4.3 Reduction to Full Normal Form

For reduction to FNF, the `Eval-f` command, like `Eval-prw`, performs a case analysis on the node tag:

- For atomic elements and boolean values, unbound logic variables, functions of non-zero arity and the empty set, no action need be taken since no further reduction can be made:

$$(\texttt{Eval-f}; C,\ a\!:\!S,\ G,\ D),\ a \xrightarrow{G} N \vdash (C,\ a\!:\!S,\ G,\ D)$$

  where

$$N \in \{\ \begin{array}{l}\mathsf{Phi},\\ \mathsf{Logic\ Var}\ a,\\ \mathsf{Atom}\ \alpha,\\ \mathsf{Bool}\ \beta,\\ \mathsf{Fun}\ n\ i,\ n > 0\quad \}\end{array}$$

- As in the other evaluation schemes, indirection nodes should be dereferenced:

$$(\texttt{Eval-f}; C,\ a\!:\!S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \vdash (\texttt{Eval-f}; C,\ a'\!:\!S,\ G,\ D)$$

- The remaining graph constructors take subexpressions, which must themselves be reduced to FNF. The general rule is:

$$(\texttt{Eval-f}; C,\ a\!:\!S,\ G,\ D),\ a \xrightarrow{G} \mathit{Node} \vdash (\mathit{CodeSequence},\ a\!:\![\,],\ G,\ \langle S, \texttt{Eval-f}; C\rangle\!:\!D)$$

  where the nodes and code sequences are according to Table 4.1.

Summary A summary of the required behaviors for the `Eval` commands may be found in Table 4.2.

4.5 Narrowing and Reducing Logic Variables

Translating the situations requiring narrowing from Table 3.1 to the new abstract machine requires several steps, in part because the model described in the table is based on a broader, more abstract notion of a graph that what is actually constructed by the machine. The model for these manipulations must be translated from the abstract graph to code/stack/graph configurations found in the actual machine. The following steps accomplish this task:

1. The situations which may require narrowing and the machine configurations which identify them must be specified.

2. New graph nodes, as specified by the model, must be introduced.

3. The introduction and manipulation of the new graph constructs must be given in terms of the abstract machine's actual components.

4. These manipulations must be implemented by augmenting the derivation relation $\vdash$.

Node	*CodeSequence*

Embed $-\tau$ e	`Push 0`
	`Retract` τ
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Return`

Pair e_1 e_2	`Push 0`
	`Left`
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Push 0`
	`Right`
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Return`

Singleton e	`Push 0`
	`SingletonElement`
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Return`

Union e_1 e_2	`Push 0`
	`UnionElements`
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Eval-wh`
	`Eval-f`
	`Pop 1`
	`Return`

Scope $-S$ e I	`_force_scoped`

Table 4.1: Code sequences for reducing nodes with subexpressions to FNF.

Node Tag	`Eval-wh`	`Eval-prw`	`Eval-f`
Atom α	continue	continue	continue
Bool β	continue	continue	continue
Embed$-\tau$ e	continue	continue	reduce subgraph to FNF
Pair e_1 e_2	continue	continue	reduce subgraphs to FNF
Fun n C, $n > 0$	continue	continue	continue
Ap e_1 e_2	save stack & code; unwind	error	error
∇ e	deref	deref	deref
Fun n C, $n = 0$	save stack & code; jump to new code	error	error
Phi	continue	continue	continue
Singleton e	continue	continue	reduce subgraph to FNF
Union e_1 e_2	continue	reduce left subchild to PRWNF; rearrange union tree if needed	reduce subgraphs to FNF
Scope$-S$ e I	continue	reduce subgraph to FNF	reduce subgraph to FNF
Diseq e_1 e_2 e'	continue	reduce subgraph to FNF	reduce subgraph to FNF

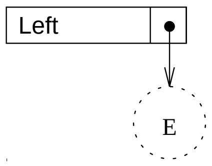
where

Continue means an expression with the given tag is already in the given form.
Deref means the subchild of the node must be taken as the node itself.
Error means that the given tag is not expected upon the execution of the given command.

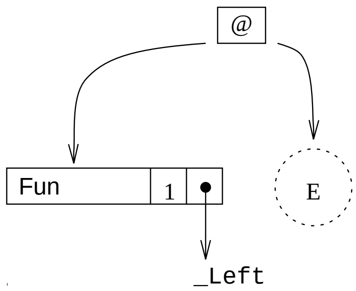
Table 4.2: The multistage evaluation commands in NG code.

4.5.1 Recognizing Cases Requiring Narrowing

In the general description of the graph model for narrowing in Chapter 3, more graph nodes are described than are actually implemented in the abstract machine. In particular, built-in functions such as **Left** and **If** are described as occupying their own nodes, *viz.*



while in the abstract machine the manipulations of the graph corresponding to these domain primitives is captured by code sequences accessed by the `Fun` node,



However, the presence of a logic variable as the right side of an application node does not by itself signal that narrowing will be necessary: many functions, especially user-defined functions, will not be strict in all their arguments. Rather, the need to narrow is more accurately signaled by the presence of a pointer to a logic variable on top of the stack at the execution time of certain abstract machine instructions. The definition of G-code instructions such as the `IsEmbedded` τ instructions or `Jfalse` illustrate this situation: these commands' definitions require knowledge about unbound quantities beyond what can be provided by just the knowledge of the set over which the variable ranges. However, as Table 3.2 shows, not all situations of these NG-code instructions applied to logic variables will require narrowing; the three cases listed there all arise from the application of the `Equals` command.

- The projection of an element of **hu** (contained within the domain **D**) into the subdomains **A**, **B** or **P**, or the determination of from which of these subdomains a value was embedded into **D**, necessarily requires more information than simply the quantity's membership in **hu**, or its inequality to certain other logic variables[1]. Thus, any application of a `Retract` τ command or an `IsEmbedded` τ command to a logic variable ranging over the Herbrand Universe[2] will require narrowing of the range of that quantifier. The machine configurations leading to this situation are

$$(\texttt{Retract}\ \tau; C,\, a{:}S,\, G,\, D)\, a \xrightarrow{G} \textsf{Logic Var}\ \ell,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Hu}\ a'\ I$$

and

$$(\texttt{IsEmbedded}\ \tau; C,\, a{:}S,\, G,\, D)\, a \xrightarrow{G} \textsf{Logic Var}\ \ell,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Hu}\ a'\ I.$$

[1] Regarding the subdomains **F** and **S**: the Herbrand Universe and each of these subdomains are disjoint, so checking for one of their tags from a logic variable ranging over **hu** will always reduce to **false**, and retracting one of their tags from a logic variable ranging over **hu** is not possible (that is, the result is the bottom element of the subdomain).

[2] The application of one of these commands to a logic variable ranging over the set of atoms or of boolean values is analogous to applying a function to a value not in its domain space, which of course maps to the bottom element of the target domain.

For the subdomains **F** and **S** the treatment is simpler: the Herbrand Universe and each of those subdomains are disjoint, so checking for one of their tags will reduce to **false** and retracting one of their tags is not possible (that is, the result is the bottom element of the subdomain).

- Two NG instructions require more information about their boolean-valued arguments than can be deduced simply from their being in **bool**, mainly `JFalse` and `Not`, in the configurations

$$(\texttt{Jfalse}\ C';C,\ a{:}S,\ G,\ D)\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Bools}\ a'\ I$$

  and

$$(\texttt{Not};C,\ a{:}S,\ G,\ D)\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Bools}\ a'\ I.$$

- Finally, the `Equals` expression gives rise to a number of cimcumstances requiring narrowing: effectively, any occurance of a logic variable as an argument to `Equals` will signal narrowing *unless current inequality information can be used to determine otherwise.*
  - If one argument of the `Equals` command is a logic variable ranging over the Herbrand Universe, then one of two types of narrowing nay be necessary:
    * If the other argument in such a case is also a logic variable ranging over **hu**, and if it cannot be determined from Diseq nodes that these two quantifiers cannot both take the same value, then the expression should narrowed into two cases,
      1. Where the two quantifiers take the same value, that is, where one is eliminated in favor of the other.
      2. Where a Diseq node is used to specify that the two nodes cannot take the same value.
    * If the other argument is an expression in the domain D, then when reduced to WHNF (as the `Equals` command would expect) it will be a domain-tagged expression.
      · If its domain tag is f or s, then since **hu** does not contain functions or sets, the entire redex can be simplified as is, without narrowing.
      · Otherwise, the range of the logic variable must be narrowed into three sets, **atoms**, **bools** and pairs of elements of **hu**, each tagged appropriately.
  - If one of the arguments is a logic variable ranging over the set of atomic elements, there are again two possible types of narrowing which may arise when there is no information about inequality relationships.
    * If the other argument is also a logic variable ranging over **atoms**, then narrowing would produce two cases like the situation of two **hu**-based quantifiers: one case where one quantifier is subsumed by the other, and one where the two are defined to be unequal.
    * If the other argument is a specific atom, narrowing would produce two cases, one where the quantifier is bound to the specific atom, and one where it is defined not to be that atom.
  - If one of the arguments is a logic variable ranging over the set of boolean values, then narrowing will always take place immediately: since the set of boolean values has only two elements, it is simplest to bind the quantifier to each of the two values in two different cases rather than creating inequality nodes which could be reduced immediately.

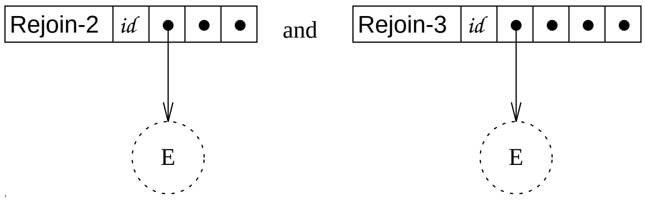
A summary of part of the discussion of the `Equals` command may be found in Table 4.3, which describes the situations requiring testing for inequalities.

Domain of logic variable	Other argument	Check Inequality Information?	Where?
hu	logic variable over **hu**	Yes	In Diseq subexpressions
	domain-tagged value	No	N/A
atoms	logic variable over **atoms**	Yes	In Diseq subexpressions
	specific atom	Yes	In Ineq list
bools	any	No	N/A

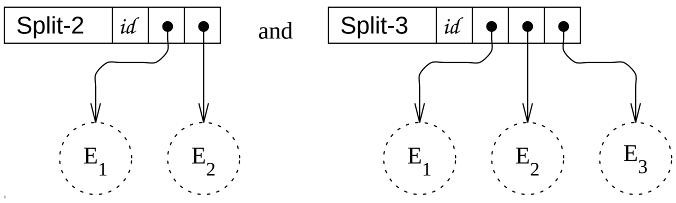
Table 4.3: Determining whether and where a check for inequality information should be made before the narrowing of values referenced by an `Equals` command.

4.5.2 Delimiters

The model for logic variables requires that distinguished graph nodes be used to delimit quasishared areas of the graph: the Rejoin$-n$ nodes mark an area of graph shared by n graph nodes, and the Split$-n$ nodes mark the end of such a shared area, and subsequently refer to n different descending subgraphs. This model can be translated directly to the graph structures of the abstract machine. Table 3.1 shows that the largest number of divergent cases which arise through narrowing is three, in the situation where a quantifier ranging over **hu** is split into three quantifiers, each one domain-tagged and ranging over **atoms**, **bools** and pairs of elements of **hu**. So four new graph nodes are required: for each of Split$-n$ and Rejoin$-n$ there could be $n = 2$ or $n = 3$, and thus the nodes:



after Rejoin$-n$, and



after Split$-n$.

As the discussions below illustrate, Rejoin nodes require some very delicate handling not relevant to Split nodes. Two different approaches to these manipulations follow: the first will be discarded in favor of the second, but is useful anyway to illustrate the motivations for the more roundabout aspects of this mechanism.

Indirections to Delimiters

The original graph model requires the Rejoin nodes to point back to the nodes set to point to it at the original narrowing time, in order that these nodes pointing to the Rejoin can be made to point to something else later, should the Rejoin node be eliminated during the upward propagation of a Split node. Since any of several fields in a node could point to a subgraph, it will be useful to insure that a Rejoin node is only ever pointed to by one specific field. If this condition is not

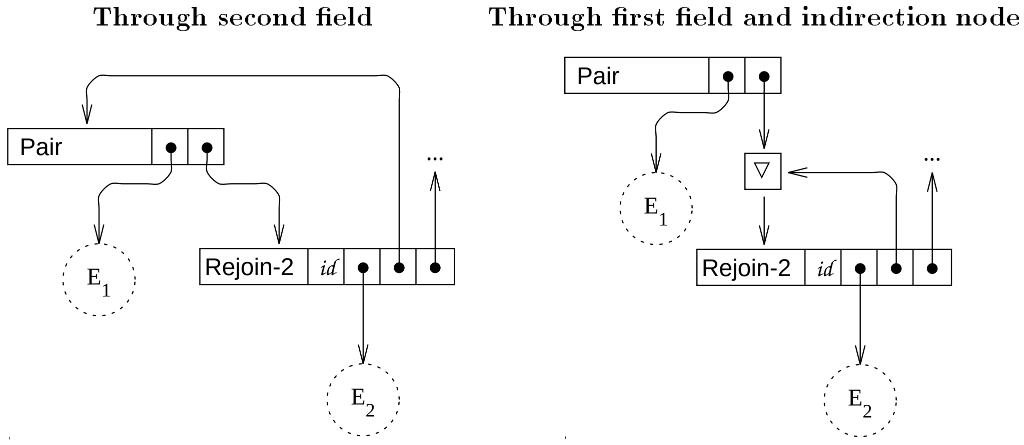


Figure 4.8: Using the first field of nodes to point to Rejoin nodes.

guaranteed, a series of pointer comparisons will have to made to find the appropriate field, which could be an expensive test. Therefore, the abstract machine will use the following simplifying assumption:

> *Any reference to a* Rejoin *node will be made through the first field of a node.*

Thus an expression accessing a Rejoin node through any field except its second one would be represented in the machine using an extra indirection node, as shown in Figure 4.8. Indirection nodes are usually removed under most garbage-collection schemes. However, an indirection node pointing to a Rejoin node which is still live in the graph must clearly *not* be removed, since doing so could violate the above assumption. Therefore, a second assumption must be made:

> *Garbage collection must not remove live indirection nodes pointing to* Rejoin *type nodes.*

These two assumptions do not apply to Split nodes, since they do not alter nodes pointing to them.

Distinguished Nodes for Delimiter Indirections

The convention of pointing to Rejoin nodes only through the first field of a node is sufficient for facilitating easy updates of the nodes pointing to the Rejoin node, but it would be very complicated to insure that such a condition be maintained. For example, if the left child of a Pair node were allowed to point to a Rejoin node, specifying rules which prevented two successive manipulations to first access that child and then build an arbitrary node with it—since the new use of the pointer may not be in the leftmost child of a node—would be exceptionally complicated, especially in light of the need not to interfere with such manipulations when they do not involve quasisharing delimiters. Even if such an enumeration were possible, it would almost certainly be of prohibitive complexity, and would add an unacceptable overhead to an actual implementation.

An increased use of indirections would not help the situation: one could expand the use of indirection node propsed above so that *all* references to Rejoin nodes are made through indirection nodes. However, indirections on the stack, and in a few other situations as well, are always dereferenced, thus reintroducing the problem of direct references to Rejoin. Worse yet, dereferencing indirections to Rejoin nodes on the stack could lead to references to Rejoin nodes which are not noted in the Rejoin node itself. The result of this situation will be that propagation of Split nodes would never alter such references (by updating the references the Rejoin node does list), leading to erroneous nontermination when such Rejoin nodes are re-encountered after propagation concludes.

A simple solution to the problem is to use another type of graph node: nodes tagged RjnPtr, for *Rejoin Pointer*, will be created at narrowing-time to be the "access points" in the graph to the Rejoin node. In addition to providing a nice resolution of the issue of accessing Rejoin nodes, RjnPtr nodes will also be quite useful in the resumption of reduction after the propagation of Split nodes (§4.5.4, p. 72).

4.5.3 Introducing Narrowing Structures into the Graph

Assembling the material of the previous two sections, the derivation relation $\vdash$ can be further extended to include narrowing. In simpler cases where narrowing will unquestionably occur, the derivation rules given here will follow a predictable pattern. The left sides of the new relation elements will be exactly those described in §4.5.1, which are exactly the cases where narrowing is required. The right sides will introduce the structures of the previous section for narrowing into the graph according to Table 3.1, which summarizes how the results of narrowing should appear.

Simple Introduction of Narrowing

This discussion parallels that of §4.5.1, which identified the configurations where narrowing would be needed.

- Identifying the subdomain from which an element has been embedded into the domain of values (that is, identifying its embedding marker) when that element ranges over **hu** requires narrowing. From Table 3.1 we have

$$\begin{array}{rr} set_1^1 = \mathbf{atoms},\ n_1 = 1 & \psi_1(x) = x_a \\ set_1^2 = \mathbf{bools},\ n_2 = 1 & \psi_2(x) = x_b \\ set_1^3 = set_2^3 = \mathbf{hu},\ n_3 = 2 & \psi_3(x,y) = \langle x, y\rangle_p \end{array}$$

  and thus the derivation is:

$$\begin{array}{l} (\texttt{IsEmbedded}\ \tau; C,\, a{:}S,\, G,\, D),\ \ \tau \in \{a,\ b,\ p\},\ \ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I \\ \vdash\ \ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G \left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' \\ a_1 \mapsto \mathsf{Embed{-}a}\ a_1' & \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 \\ a_1' \mapsto \mathsf{Logic\ Var}\ \ell_1 & \ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I \\ a_2 \mapsto \mathsf{Embed{-}b}\ a_2' & \ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I \\ a_2' \mapsto \mathsf{Logic\ Var}\ \ell_2 & \ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I \\ a_3 \mapsto \mathsf{Embed{-}p}\ a_3' & \ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I \\ a_3' \mapsto \mathsf{Pair}\ r_1\ r_2 & r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 \\ I \mapsto \mathsf{Ineqs{-}End} & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\ p_1 \mapsto \mathsf{RjnPtr}\ e\ \ \ p_2 \mapsto \mathsf{RjnPtr}\ e & p_3 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3 & \\ a \mapsto \mathsf{Split}\ 3\ id\ a_1\ a_2\ a_3 & id \mapsto \mathsf{UID} \end{array}\right],\ D) \end{array}$$

  A graphical representation of this manipulation is shown in Figure 4.9. There is no more reference made to what was originally the subsequent NG code to be executed, since the effect is that the construction and reduction of the graph is rewound to an earlier point, with the intention of proceeding from that point with new information. The upward propagation of the Split node after narrowing is performed by the `PropSplitUp` command, which we use without definition in this section and introduce formally in §4.5.4, p. 72. We have included a restriction on the tags for this case because an element in the Herbrand Universe *cannot* be a function or a set: thus we also have the simple rule

$$\begin{array}{l} (\texttt{IsEmbedded}\ \tau; C,\, a{:}S,\, G,\, D),\ \ \tau \in \{f,\ s\},\ \ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I \\ \ \vdash\ \ (C,\, a'{:}S,\, G\,[a' \mapsto \mathsf{Bool\ false}]\,,\, D) \end{array}$$

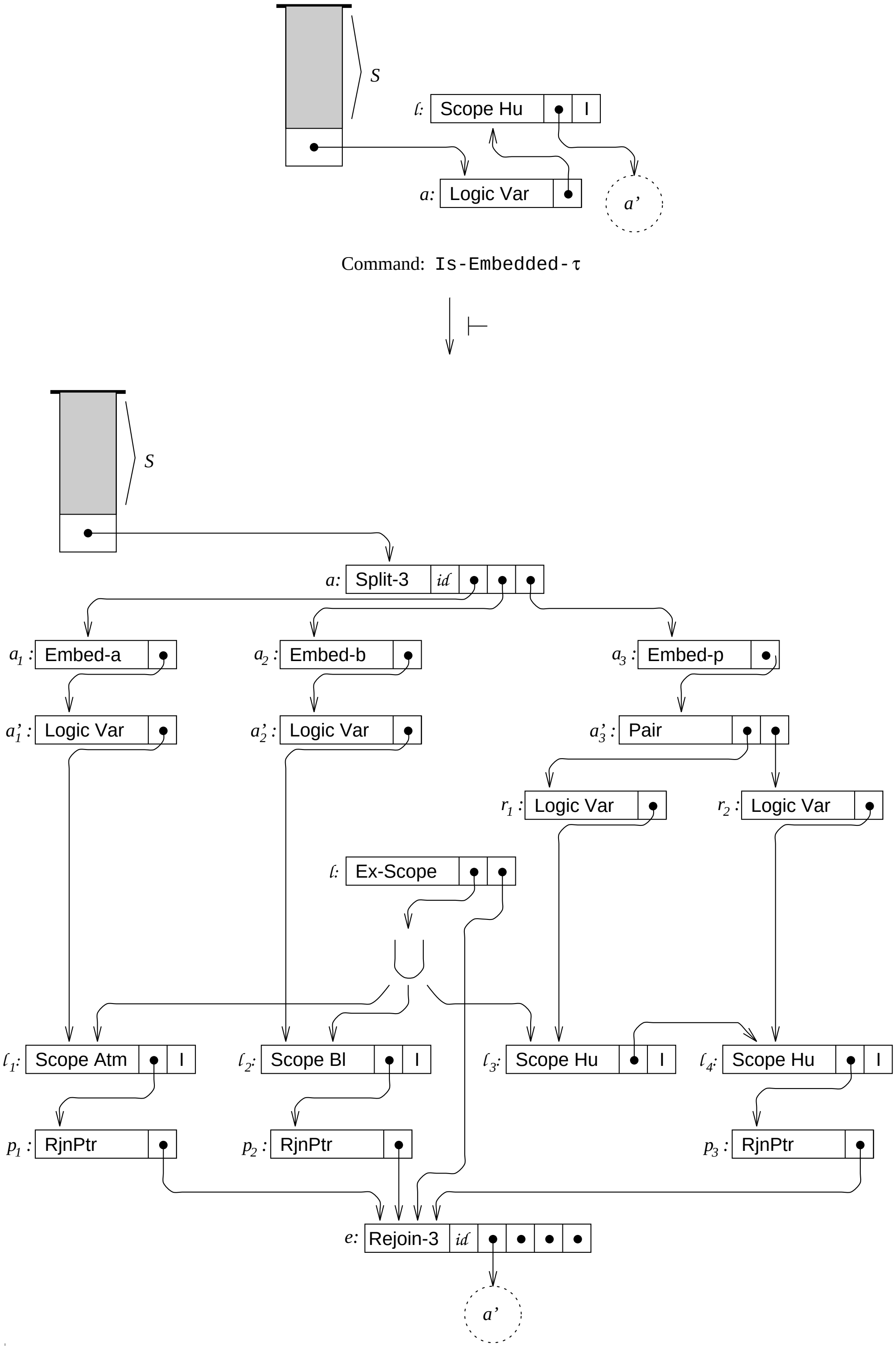


Figure 4.9: Illustrating the derivation rule introducing narrowing for the `IsEmbedded` τ command.

The rule for $\mathsf{Retract}{-}\tau$ is similar to the one above:

$$
\begin{array}{l}
(\texttt{Retract}\ \tau; C,\ a{:}S,\ G,\ D)\ \ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I \\
\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G \left[\begin{array}{ll}
\ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' \\
a_1 \mapsto \mathsf{Embed{-}a}\ a'_1 & \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 \\
a'_1 \mapsto \mathsf{Logic\ Var}\ \ell_1 & \ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I \\
a_2 \mapsto \mathsf{Embed{-}b}\ a'_2 & \ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I \\
a'_2 \mapsto \mathsf{Logic\ Var}\ \ell_2 & \ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I \\
a_3 \mapsto \mathsf{Embed{-}p}\ a'_3 & \ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I \\
a'_3 \mapsto \mathsf{Pair}\ r_1\ r_2 & r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 \\
I \mapsto \mathsf{Ineqs{-}End} & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\
\multicolumn{2}{l}{p_1 \mapsto \mathsf{RjnPtr}\ e\quad p_2 \mapsto \mathsf{RjnPtr}\ e\quad p_3 \mapsto \mathsf{RjnPtr}\ e} \\
\multicolumn{2}{l}{e \mapsto \mathsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3} \\
a \mapsto \mathsf{Split}\ 3\ id\ a_1\ a_2\ a_3 & id \mapsto \mathsf{UID}
\end{array} \right],\ D)
\end{array}
$$

- The execution of the `JFalse`, `Not` or `And` commands on a logic variable will require immediate narrowing of the quantifier. The rule scheme for these commands is:

$$
\begin{array}{l}
(X; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I \\
\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G \left[\begin{array}{ll}
\ell' \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell_2 \\
a_1 \mapsto \mathsf{Bool}\ \mathbf{true} & p_1 \mapsto \mathsf{RjnPtr}\ e \\
a_2 \mapsto \mathsf{Bool}\ \mathbf{false} & p_2 \mapsto \mathsf{RjnPtr}\ e \\
\multicolumn{2}{l}{e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2} \\
a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \mathsf{UID}
\end{array} \right],\ D)
\end{array}
$$

where $X = $ `JFalse`, $X = $ `Not` or $X = $ `And`.

- Certain tests for equality involving logic variables will not require examination of inequality lists, as shown in Table 4.3.

  – Comparing a logic variable ranging over the Herbrand Universe with an item marked as embedded will require narrowing so that the subsequent quantifiers will themselves have an embedding marker for comparison. The cases for narrowing are the same as for `IsEmbedded` τ, and the derivation rules are:

$$
\begin{array}{l}
(\texttt{Equals}; C,\ a{:}b{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed{-}}\tau\ a',\ b \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ b'\ I \\
\vdash\ (\texttt{PropSplitUp}\ id\ b,\ a{:}b{:}S,\ G \left[\begin{array}{ll}
\ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & p_1 \mapsto \mathsf{RjnPtr}\ e \\
\ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' & p_2 \mapsto \mathsf{RjnPtr}\ e \\
\ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 & p_3 \mapsto \mathsf{RjnPtr}\ e \\
b_1 \mapsto \mathsf{Embed{-}a}\ b'_1 & b'_1 \mapsto \mathsf{Logic\ Var}\ \ell_1 \\
b_2 \mapsto \mathsf{Embed{-}b}\ b'_2 & b'_2 \mapsto \mathsf{Logic\ Var}\ \ell_2 \\
b_3 \mapsto \mathsf{Embed{-}p}\ b'_3 & b'_3 \mapsto \mathsf{Pair}\ r_1\ r_2 \\
r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\
\multicolumn{2}{l}{\ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I} \\
\multicolumn{2}{l}{\ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I} \\
\multicolumn{2}{l}{\ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I\quad I \mapsto \mathsf{Ineqs{-}End}} \\
\multicolumn{2}{l}{\ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I\quad id \mapsto \mathsf{UID}} \\
\multicolumn{2}{l}{e \mapsto \mathsf{Rejoin}\ 3\ id\ b'\ p_1\ p_2\ p_3} \\
\multicolumn{2}{l}{b \mapsto \mathsf{Split}\ 3\ id\ b_1\ b_2\ b_3}
\end{array} \right],\ D)
\end{array}
$$

$$(\texttt{Equals}; C,\, a{:}b{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Logic Var}\ \ell,\ b \xrightarrow{G} \textsf{Embed}{-}\tau\ b',\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Hu}\ a'\ I$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}b{:}S,\ G \left[\begin{array}{ll} \ell \mapsto \textsf{Ex-Scope}\ \ell'\ e & p_1 \mapsto \textsf{RjnPtr}\ e \\ \ell' \mapsto \textsf{Union}\ \ell_1\ \ell'' & p_2 \mapsto \textsf{RjnPtr}\ e \\ \ell'' \mapsto \textsf{Union}\ \ell_2\ \ell_3 & p_3 \mapsto \textsf{RjnPtr}\ e \\ a_1 \mapsto \textsf{Embed}{-}\textsf{a}\ a_1' & a_1' \mapsto \textsf{Logic Var}\ \ell_1 \\ a_2 \mapsto \textsf{Embed}{-}\textsf{b}\ a_2' & a_2' \mapsto \textsf{Logic Var}\ \ell_2 \\ a_3 \mapsto \textsf{Embed}{-}\textsf{p}\ a_3' & a_3' \mapsto \textsf{Pair}\ r_1\ r_2 \\ r_1 \mapsto \textsf{Logic Var}\ \ell_3 & r_2 \mapsto \textsf{Logic Var}\ \ell_4 \\ \ell_1 \mapsto \textsf{Scope}{-}\textsf{Atoms}\ p_1\ I & \\ \ell_2 \mapsto \textsf{Scope}{-}\textsf{Bools}\ p_2\ I & \\ \ell_3 \mapsto \textsf{Scope}{-}\textsf{Hu}\ \ell_4\ I \quad id \mapsto \textsf{UID} & \\ \ell_4 \mapsto \textsf{Scope}{-}\textsf{Hu}\ p_3\ I \quad I \mapsto \textsf{Ineqs}{-}\textsf{End} & \\ e \mapsto \textsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3 & \\ a \mapsto \textsf{Split}\ 3\ id\ a_1\ a_2\ a_3 & \end{array}\right],\ D)$$

– Any test for equality involving logic variables ranging over boolean values requires that narrowing be performed for that quantifier:

$$(\texttt{Equals}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Logic Var}\ \ell,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Bools}\ a'\ I$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G \left[\begin{array}{ll} \ell \mapsto \textsf{Ex-Scope}\ \ell'\ e & \ell' \mapsto \textsf{Union}\ \ell_1\ \ell_2 \\ a_1 \mapsto \textsf{Bool}\ \mathbf{true} & p_1 \mapsto \textsf{RjnPtr}\ e \\ a_2 \mapsto \textsf{Bool}\ \mathbf{false} & p_2 \mapsto \textsf{RjnPtr}\ e \\ e \mapsto \textsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \textsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \textsf{UID} \end{array}\right],\ D)$$

Situations Requiring Queries into the Graph

The remaining situations which may require narrowing first require an examination of known inequality information. Again making reference to Table 4.3, one can see that depending on the variety of items under scrutiny, this search will be conducted in only one of the two locations where inequality information is stored. Searching a list of Ineq nodes is simply described. Two basic configurations will initiate the manipulation:

$$(\texttt{Equals}{:}\, C,\, a{:}b{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Logic Var}\ \ell,\ b \xrightarrow{G} \textsf{Atom}\ \alpha,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Atoms}\ a'\ I$$
$$\vdash\ (\texttt{Search-Ineq}\ \alpha\ a,\ I{:}[\,],\ G,\ \langle a{:}b{:}S, C\rangle{:}\,D)$$

$$(\texttt{Equals}{:}\, C,\, a{:}b{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Atom}\ \alpha,\ b \xrightarrow{G} \textsf{Logic Var}\ \ell,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Atoms}\ b'\ I$$
$$\vdash\ (\texttt{Search-Ineq}\ \alpha\ b,\ I{:}[\,],\ G,\ \langle a{:}b{:}S, C\rangle{:}\,D)$$

The list should be traversed until either it is exhausted or a match is found. Finding the atom in the list means that the result of the comparison should be **false**, and that no narrowing is necessary:

$$(\texttt{Search-Ineq}\ \alpha\ u,\ I{:}[\,],\ G,\ \langle a_1{:}a_2{:}S, C\rangle{:}\,D),\ I \xrightarrow{G} \textsf{Ineq}\ \alpha\ I'\ \vdash\ (C,\ a{:}S,\ G[a \mapsto \textsf{Bool}\ \mathit{false}],\ D)$$

$$(\texttt{Search-Ineq}\ \alpha\ u,\ I{:}[\,],\ G,\ D),\ I \xrightarrow{G} \textsf{Ineq}\ \alpha'\ I',\ \alpha \neq \alpha'\ \vdash\ (\texttt{Search-Ineq}\ \alpha\ u,\ I'{:}[\,],\ G,\ D)$$

Exhausting the list of inequalities means that narrowing *will* be necessary, since the available information does not allow a reduction:

$$(\texttt{Search-Ineq}\ \alpha\ u,\ I_0{:}[\,],\ G,\ \langle S, C\rangle{:}\,D),\ I_0 \xrightarrow{G} \textsf{Ineqs}{-}\textsf{End},$$
$$u \xrightarrow{G} \textsf{Logic Var}\ \ell,\ \ell \xrightarrow{G} \textsf{Scope}{-}\textsf{Atoms}\ u'\ I$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ u,\ S,\ G \left[\begin{array}{ll} \ell \mapsto \textsf{Ex-Scope}\ \ell'\ e & I' \mapsto \textsf{Ineq}\ \alpha\ I \\ \ell' \mapsto \textsf{Union}\ p_1\ \ell_2 & u_1 \mapsto \textsf{Atom}\ \alpha \\ \ell_2 \mapsto \textsf{Scope}{-}\textsf{Atoms}\ p_2\ I' & u_2 \mapsto \textsf{Logic Var}\ \ell_2 \\ p_1 \mapsto \textsf{RjnPtr}\ e & p_2 \mapsto \textsf{RjnPtr}\ e \\ e \mapsto \textsf{Rejoin}\ 2\ id\ u'\ p_1\ p_2 & \\ u \mapsto \textsf{Split}\ 2\ id\ u_1\ u_2 & id \mapsto \textsf{UID} \end{array}\right],\ D)$$

Accessing the information stored in Diseq nodes is more complicated. These nodes are found at the base of stacks stored on the dump. The dump is just a stack (albeit specialized), so one might access this information in the manner natural for stacks, namely, by popping the stack and examining the top element. But simply popping these stack/code pairs from the dump would also *lose* those pairs, and if it turns out that narrowing is *not* to be performed, then these stored pairs will be needed as a part of ordinary (G-machine) reduction. Thus we will use a more sophisticated mechanism to examine the dump: by encoding an offset into the dump in the NG-code.

There are two cases where inequality information in Diseq nodes must be accessed: the comparison of logic variables ranging over atomic elements, and of logic variables ranging over the Herbrand universe. In fact, these can both be described under the same mechanism. The comparison is again started by the `Equals` command:

$$\begin{array}{l}(\texttt{Equals}; C,\ a_1{:}a_2{:}\ldots{:}a_n,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2 \\ \quad\vdash\ (\texttt{Search-Diseq}\ \ell_1\ \ell_2\ 1,\ a_n{:}[\,],\ G,\ \langle a_1{:}\ldots{:}a_n, C\rangle{:}D)\end{array}$$

While searching through the stacks stored on the dump, each step will result in one of the following:

1. the node atop the stack will declare the scope of one of the logic variables: at this point narrowing should occur, since no relevant disequality information would have been found.

$$\begin{array}{l}(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ \ell_1{:}S_0,\ G,\ D),\ \ell_1 \xrightarrow{G} \mathsf{Scope{-}}S\ a'\ I,\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1 \\ \quad D = \langle a{:}b{:}S, C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle s_1{:}\ldots{:}s_m{:}\ell_1{:}[\,], C_0\rangle{:}D'' \\ \vdash\ (\texttt{PropSplitUp}\ id\ \ell_1,\ a{:}b{:}S,\ G\left[\begin{array}{ll}\ell_1 \mapsto \mathsf{Ex{-}Scope}\ \ell_1'\ \ell' & \ell_1' \mapsto \mathsf{Union}\ p_1\ \ell_b \\ a_1 \mapsto \mathsf{Logic\ Var}\ \ell_2 & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Logic\ Var}\ \ell_b & \ell_b \mapsto \mathsf{Scope{-}}S\ \ell_b'\ I \\ id \mapsto \mathsf{UID} & \ell_b' \mapsto \mathsf{Diseq}\ \ell_b\ \ell_2\ p_2 \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & p_2 \mapsto \mathsf{RjnPtr}\ e\end{array}\right],\ D')\end{array}$$

$$\begin{array}{l}(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ \ell_2{:}S_0,\ G,\ D),\ \ell_2 \xrightarrow{G} \mathsf{Scope{-}}S\ a\ I \\ \quad D = \langle a{:}b{:}S, C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle s_1{:}\ldots{:}s_m{:}\ell_2{:}[\,], C_0\rangle{:}D'' \\ \vdash\ (\texttt{PropSplitUp}\ id\ \ell_2,\ a{:}b{:}S,\ G\left[\begin{array}{ll}\ell_2 \mapsto \mathsf{Ex{-}Scope}\ \ell_2'\ \ell' & \ell_2' \mapsto \mathsf{Union}\ p_1\ \ell_b \\ a_1 \mapsto \mathsf{Logic\ Var}\ \ell_1 & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Logic\ Var}\ \ell_b & \ell_b \mapsto \mathsf{Scope{-}}S\ \ell_b'\ I \\ id \mapsto \mathsf{UID} & \ell_b' \mapsto \mathsf{Diseq}\ \ell_b\ \ell_1\ p_2 \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & p_2 \mapsto \mathsf{RjnPtr}\ e\end{array}\right],\ D')\end{array}$$

2. the node atop the stack is a disequality node establishing that the two nodes under consideration are not equal. Reduction should then proceed from the code and stack stored on top of the dump, altering the stack to reflect that the result of the comparison was **false**.

$$\begin{array}{l}(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ S_0,\ G,\ D), \\ \quad D = \langle a_1{:}a_2{:}S, C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle a{:}[\,], C_0\rangle{:}D'' \\ \quad a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a',\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2 \\ \vdash\ (C,\ b{:}S,\ G[b \mapsto \mathsf{Bool\ false}],\ D) \\ (\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ S_0,\ G,\ D), \\ \quad D = \langle a_2{:}a_1{:}S, C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle a{:}[\,], C_0\rangle{:}D'' \\ \quad a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a',\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2 \\ \vdash\ (C,\ b{:}S,\ G[b \mapsto \mathsf{Bool\ false}],\ D)\end{array}$$

3. the node atop the stack is neither of the above two. In this case the last element of the stack above the one now under consideration should be pushed from the dump to the stack.

$$
\begin{array}{l}
(\texttt{Search-Diseq}\,\ell_1\ \ell_2\ n,\, S,\, G,\, D),\ \ D = d_0{:}\ldots{:}\,d_{n-1}{:}\,\langle a_0{:}\ldots a_j{:}\,[\,], C_0\rangle{:}\,D', \\
\quad a_j \xrightarrow{G} N,\ N \notin \ \{\mathsf{Logic\ Var}\ \ell_1, \\
\qquad\qquad\qquad\quad \mathsf{Logic\ Var}\ \ell_2, \\
\qquad\qquad\qquad\quad \mathsf{Diseq}\ a_1\ a_2\ a', \\
\qquad\qquad\qquad\quad \mathsf{Diseq}\ a_2\ a_1\ a' \quad \} \\
\ \vdash\ (\texttt{Search-Diseq}\,\ell_1\ \ell_2\ (n+1),\, S,\, G,\, D)
\end{array}
$$

4.5.4 Quasisharing Delimiters and Derivation

This section describes the interaction of quasisharing delimiters with the standard commands, and the manipulation of the graph performed by commands such as `SplitPropUp`.

Evaluating and Unwinding Split Nodes

Since subexpressions are always reduced to WHNF before they are deconstructed (say with the `Left` command) or reduced to a stronger normal form, encounters with Split nodes will always be made by either the `Eval` or the `Unwind` command. The result of such an encounter should be the same as when narrowing is initiated, namely, upward propagation is required:

$$
\begin{array}{rl}
(\texttt{Eval-wh};\, C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n & \vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}\,S,\, G,\, D) \\
(\texttt{Eval-prw};\, C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n & \vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}\,S,\, G,\, D) \\
(\texttt{Eval-f};\, C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n & \vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}\,S,\, G,\, D) \\
(\texttt{Unwind};\, C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n & \vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}\,S,\, G,\, D)
\end{array}
$$

Propagating Split Nodes

The `PropSplitUp` command indicates that unsharing of graph nodes is currently in progress. When propagation upwards through stack configurations held on the dump reaches the stack of one element pointing to the node which *used* to be the scope delimiter corresponding to this narrowing, propagation should end. The Split and Rejoin nodes can be eliminated, and the marker for the top of the old scope delimiters made into an ordinary indirection node. Again the calling configuration of §4.1 is used to re-execute the command which initiated the reduction sequence leading to the just-completed propagation.

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ s,\, a{:}\,[\,],\, G,\ \langle S, C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{Ex\text{-}Scope}\ a'\ r, \\
\quad r \xrightarrow{G} \mathsf{Rejoin}\ 2\ id\ s\ r_1\ r_2,\ s \xrightarrow{G} \mathsf{Split}\ 2\ id\ s_1\ s_2, \\
\quad r_1 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ r \\
\ \vdash\ (C,\, a'{:}\,S,\, G \begin{bmatrix} a \mapsto \nabla\ a' \\ r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \end{bmatrix},\, D)
\end{array}
$$

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ s,\, a{:}\,[\,],\, G,\ \langle S, C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{Ex\text{-}Scope}\ a'\ r, \\
\quad r \xrightarrow{G} \mathsf{Rejoin}\ 3\ id\ s\ r_1\ r_2\ r_3,\ s \xrightarrow{G} \mathsf{Split}\ 3\ id\ s_1\ s_2\ s_3, \\
\quad r_1 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_3 \xrightarrow{G} \mathsf{RjnPtr}\ r \\
\ \vdash\ (C,\, a{:}\,S,\, G \begin{bmatrix} r_1 \mapsto \nabla\ s_1 & r_2 \mapsto \nabla\ s_2 \\ r_3 \mapsto \nabla\ s_3 & a \mapsto \nabla\ a' \end{bmatrix},\, D)
\end{array}
$$

While the initial propagation after a narrowing is stopped by the Ex-Scope node, subsequent propagation of Split nodes associated with the same narrowing would be terminated by a Rejoin node:

$$
\begin{aligned}
&(\texttt{PropSplitUp}\ id\ a',\ a\!:\![\,],\ G,\ \langle S, C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{Rejoin}\ 2\ id\ a'\ r_1\ r_2,\\
&\qquad a' \xrightarrow{G} \mathsf{Split}\ 2\ id\ s_1\ s_2,\ r_1 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ a\\
&\vdash\ \ (C,\ S,\ G\begin{bmatrix} r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \end{bmatrix},\ D)\\
&(\texttt{PropSplitUp}\ id\ a',\ a\!:\![\,],\ G,\ \langle S, C\rangle\!:D),\\
&\qquad a \xrightarrow{G} \mathsf{Rejoin}\ 3\ id\ a'\ r_1\ r_2\ r_3,\ a' \xrightarrow{G} \mathsf{Split}\ 3\ id\ s_1\ s_2\ s_3,\\
&\qquad r_1 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_3 \xrightarrow{G} \mathsf{RjnPtr}\ a\\
&\vdash\ \ (C,\ S,\ G\begin{bmatrix} r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \\ r_3 \mapsto \nabla\ s_3 \end{bmatrix},\ D)
\end{aligned}
$$

The above four rules for terminating propagation rely on two previous design decisions:

1. As mentioned, the calling configuration of §4.1 allows a neat re-execution of the command originally run on the just-narrowed expression. That is, if a call to the NG-code instruction c caused a code/stack pair to be stored on the dump, that pair would be of the form

$$\langle S, c; C\rangle\,,$$

where C is the list of instructions originally following c and S was the configuration of the stack at the time of execution of c.

2. The use of the RjnPtr nodes, rather than ordinary indirection nodes of the first-field convention, allows easy updating, and more importantly, an easy transaction back to reduction: if only RjnPtr nodes and never the Rejoin nodes themselves are addressed on the stack as a result of normal reduction, then no investigation of the internals of the stacks on the dump must be performed. Were Rejoin nodes addressable from spine and argument stacks, it would be necessary to discard all stacks on the dump referencing the Rejoin node: although indirections to Rejoin nodes are automatically updated, a direct reference on the stack would be useless after propagation since *only* indirect references are updated.

Of course, manipulations of Rejoin and RjnPtr nodes must set up the stack and dump configurations these rules expect to find (p. 77).

Actual duplication of the graph occurs when the Split is below some other sort of node. The specific manipulation depends on the current stack configuration.

- When a Split node is one of the ribs of a reduction spine which has been unwound to find a Fun node and sufficient arguments, and then reconfigured to execute the function, the structure of the spine should be duplicated. The question of how the other ribs of the spine should be treated remains:
  - Those ribs whose topmost node is *not* a Split node of the same *id* as the one we are currently propagating should *not* be duplicated.

    That is, if the outermost node of the rib is a quasisharing delimiter of some other *id*, or if it is some other type of node, we may never have to duplicate that graph. Unless and until we are required to evaluate it, and in the course of that evaluation find a Split node, we shall assume that that rib is shared by all of the spines to be produced, in which case its duplication would be wasteful.

    In this case, we place the rib to be shared under a Rejoin node. Intuitively, we have several independant section of graph "above," but these are "rejoined" into a single quasishared section of graph "below".

- Alternatively, the top node of the rib could be a Split node identified by the same unique id as the one we are manipulating. The n children of that Split node would then correspond to the n spines which we are creating. So the first spine would point to the first of the Split node's children; the second spine, to the second of its children; and so on.

One could give a single rule to reduce the entire spine:

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ a_k,\ x_0{:}\ldots{:}x_m{:}a_0{:}\ldots{:}a_n{:}e_n{:}[\,],\ G,\ \langle S,C\rangle{:}D),\\
\qquad \text{where}\quad e_i \xrightarrow{G} \mathsf{Ap}\ e_{i-1}\ a_{i-1},\ 1 \le i \le n,\\
\qquad\qquad\quad\ a_k \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ \text{constant}\ k\\
\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\left[\begin{array}{l}
e_i \mapsto \mathsf{Split}\ 2\ id\ e'_i\ e''_i,\ 1 \le i \le n\\
e'_i \mapsto \mathsf{Ap}\ e'_{i-1}\ a'_{i-1},\ 1 \le i \le k,\ k+1 < i \le n\\
e''_i \mapsto \mathsf{Ap}\ e''_{i-1}\ a''_{i-1},\ 1 \le i \le k,\ k+1 < i \le n\\
e'_{k+1} \mapsto \mathsf{Ap}\ e'_k\ a' \quad a'_i \mapsto \nabla\ r'_i,\ i \ne k\\
e''_{k+1} \mapsto \mathsf{Ap}\ e''_k\ a'' \quad a''_i \mapsto \nabla\ r''_i,\ i \ne k\\
r'_i \mapsto \mathsf{Rejoin}\ 2\ id\ a_i\ a'_i\ a''_i,\ i \ne k
\end{array}\right],\ D)
\end{array}
$$

However, this rule is inadequate because it fails to account for a number of possibilities:

1. There may be more than one application node whose right-hand side points to the Split node being propagated. In this case, the rule given here is not deterministic, since either of those application nodes could be taken as e_{k+1} above, and moreover, it is wasteful of heap space, since for the remainder of the nodes pointing to the same Split node, those not taken as e_{k+1}, pairs of Rejoin and Split nodes are created which would be immediately removed upon reduction.
2. Some or all of the Split nodes at the tops of ribs may be behind indirection nodes.

These problems can be remedied, and an altogether simpler scheme defined, by propagating the Split node up the spine one step at a time. The first step in manipulating the spine is to determine how many of the pointers on the stack are to arguments, as opposed to pointers to other intermediate structures created by NG code.

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ e,\ x_0{:}\ldots{:}x_m{:}a_1{:}\ldots{:}a_n{:}e_n{:}[\,],\ G,\ D), e_0 \xrightarrow{G} \mathsf{Fun}\ n\ C\\
\quad e_i \xrightarrow{G} \mathsf{Ap}\ e_{i-1}\ a_n,\ 1 \le i \le n\\
\quad\vdash\ (\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D)
\end{array}
$$

The initial duplication, just above the Fun node, must be distinguished because the Fun node has not been duplicated, while at other points in the spine the left child will have been duplicated. Proceeding, the `InitSpineSplit` command is defined as follows.

- Any indirection nodes covering the right subchild of the node pointed to from the top of the stack (the lowest application node not yet processed) must be removed:

$$
\begin{array}{l}
(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,\ a \xrightarrow{G} \nabla\ a'\\
\quad\vdash\ (\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a'],\ D)
\end{array}
$$

- If the right subchild is a Split node identified by id, then the application node can be duplicated, with each duplicate receiving one of the n children of the Split node:

$$
\begin{array}{l}
(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,\\
\quad a \xrightarrow{G} \mathsf{Split}\ m\ id\ s_1\ \ldots\ s_m\\
\quad\vdash\ (\texttt{SpineSplit}\ id\ e_1,\ e_2{:}\ldots{:}e_n{:}[\,],\ G\left[\begin{array}{l}
e_1 \mapsto \mathsf{Split}\ m\ id\ e'_1\ \ldots\ e'_m\\
e'_i \mapsto \mathsf{Ap}\ f\ s_i,\ 1 \le i \le m
\end{array}\right],\ D)
\end{array}
$$

– Otherwise, an indirection to a new Rejoin node must be the right child of the new application nodes:

$$(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
a is a pointer neither to a Split node identified by id, nor to an indirection node
$$\vdash\ (\texttt{SpineSplit}\ id\ e_1,\ e_2{:}\ldots{:}e_n{:}[\,],\ G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_m \\ e'_i \mapsto \mathsf{Ap}\ f\ r_i,\ 1 \le i \le m \\ r_i \mapsto \mathsf{RjnPtr}\ r,\ 1 \le i \le m \\ r \mapsto \mathsf{Rejoin}\ n\ id\ a\ r_1\ \ldots\ r_m \end{array}\right],\ D)$$

The rules for the SpineSplit command are very similar to those for InitSpineSplit, except that the left child of the application node pointed to from the top of the stack is assumed to be either a Split node identified with id, or an indirection to one.

$$(\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,\ \ f \xrightarrow{G} \nabla\ f'$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_1 \xrightarrow{G} \mathsf{Ap}\ f'\ a'],\ D)$$
$$(\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ e_0\ a,\ \ a \xrightarrow{G} \nabla\ a',$$
$$e_0 \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_m$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_1 \xrightarrow{G} \mathsf{Ap}\ e_0\ a'],\ D)$$
$$(\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_m{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
$$f \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_n,\ \ a \xrightarrow{G} \mathsf{Split}\ n\ id\ s_1\ \ldots\ s_n$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_1,\ e_2{:}\ldots{:}e_m{:}[\,],\ G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_n \\ e'_i \mapsto \mathsf{Ap}\ f_i\ s_i,\ 1 \le i \le n \end{array}\right],\ D)$$
$$(\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_m{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
$$f \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_n,$$
a is a pointer neither to a Split node identified by id, nor to an indirection node unless that indirection is to a Rejoin node
$$\vdash\ (\texttt{SpineSplit}\ id\ e_1,\ e_2{:}\ldots{:}e_m{:}[\,],\ G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_n \\ e'_i \mapsto \mathsf{Ap}\ f_i\ r_i,\ 1 \le i \le n \\ r_i \mapsto \mathsf{RjnPtr}\ r,\ 1 \le i \le n \\ r \mapsto \mathsf{Rejoin}\ n\ id\ a\ r_1\ \ldots\ r_n \end{array}\right],\ D)$$

Duplication of the spine terminates when the local stack is emptied. Then, propagation of the Split node upwards can continue with the next stack stored on the dump:

$$(\texttt{SpineSplit}\ id\ e_n,\ [\,],\ G,\ \langle S, C\rangle{:}D)\ \vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G,\ D)$$

- Another case similar to the above is that of spines of application nodes, rather than their arguments, on the dump. The latter case would be formed from one of these, where the result would be a function or partial application which would then be returned to the sequence of application nodes and re-unwound. Since the lowest vertebrate of this spine will point to the same element as the base of the stack which had been atop this one in the dump, that node will have been updated with a Split node by the time the stack in question is removed from the dump. This is exactly the configuration established after a duplication made by the InitSpineSplit command, and so that command can also be used in this context:

$$(\texttt{PropSplitUp}\ id\ e_0,\ e_1{:}\ldots{:}e_n,\ G,\ D),\ \ e_i \xrightarrow{G} \mathsf{Ap}\ e_{i-1}\ a_i$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n,\ G,\ D)$$

- In the cases above, the presence of a spine at the base of a stack taken from the dump during propagation means that the particular manipulation leading to this narrowing was triggered by the `Eval-wh` command. Since the `Eval-prw` and `Eval-f` commands can cause reductions within the body of expressions already in WHNF, data constructor nodes can also be referenced from the bottom of stacks. For example, a Split node could be propagated upwards through either side of a Pair node:

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ a_1,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G,\ \langle S,C\rangle\!:\!D),\\
\quad e_n \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2,\ a_1 \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ a_1 \neq a_2,\ a_2 \text{ not an indirection}\\
\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\left[\begin{array}{ll} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' & \\ e' \mapsto \mathsf{Pair}\ a'\ r_1 & r_1 \mapsto \mathsf{RjnPtr}\ r \\ e'' \mapsto \mathsf{Pair}\ a''\ r_2 & r_2 \mapsto \mathsf{RjnPtr}\ r \\ r \mapsto \mathsf{Rejoin}\ 2\ id\ a_2\ r_1\ r_2 & \end{array}\right],\ D)\\
(\texttt{PropSplitUp}\ id\ a_2,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G,\ \langle S,C\rangle\!:\!D),\\
\quad e_n \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2,\ a_2 \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ a_1 \neq a_2,\\
\quad a_1 \text{ not an indirection}\\
\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\left[\begin{array}{ll} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' & \\ e' \mapsto \mathsf{Pair}\ r_1\ a' & r_1 \mapsto \mathsf{RjnPtr}\ r \\ e'' \mapsto \mathsf{Pair}\ r_2\ a'' & r_2 \mapsto \mathsf{RjnPtr}\ r \\ r \mapsto \mathsf{Rejoin}\ 2\ id\ a_1\ r_1\ r_2 & \end{array}\right],\ D)
\end{array}
$$

If either side of the pair is an indirection node, the indirection should first be dereferenced in order to find all Split nodes.

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ e,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G,\ D),\ e_n \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2, a_1 \xrightarrow{G} \nabla\ a'\\
\quad\vdash\ (\texttt{PropSplitUp}\ id\ e,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G[e_n \mapsto \mathsf{Pair}\ a'\ a_2],\ D)\\
(\texttt{PropSplitUp}\ id\ e,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G,\ D),\ e_n \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2,\\
\quad\quad a_1 \text{ not an indirection},\ a_2 \xrightarrow{G} \nabla\ a'\\
\quad\vdash\ (\texttt{PropSplitUp}\ id\ e,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G[e_n \mapsto \mathsf{Pair}\ a_1\ a'],\ D)
\end{array}
$$

Conceivably, both sides of a Pair node could be propagated simultaneously:

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ a,\ e_1\!:\!\ldots\!:\!e_n\!:\![\,],\ G,\ \langle S,C\rangle\!:\!D),\\
\quad a \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ e_n \xrightarrow{G} \mathsf{Pair}\ a\ a\\
\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\left[\begin{array}{l} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' \\ e' \mapsto \mathsf{Pair}\ a'\ a' \\ e'' \mapsto \mathsf{Pair}\ a''\ a'' \end{array}\right],\ D)
\end{array}
$$

Other nodes with more than one child are handled like the Pair node and for the sake of brevity are not enumerated here. The interaction of upward propagation of Split nodes with these nodes in described in the Appendix A.

- Less complicated than nodes such as Pair, which have two children, are the data constructors with only one child. A similar process is used to duplicate these nodes, but since no Rejoin nodes must be farmed out over the other children the descriptions are shorter. For example, the rule schema for the Embed$-\tau$ nodes are:

$$
\begin{array}{l}
(\texttt{PropSplitUp}\ id\ a',\ x_0\!:\!\ldots\!:\!x_n\!:\!a\!:\![\,],\ G,\ \langle S,C\rangle\!:\!D),\\
\quad a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a',\ a' \xrightarrow{G} \mathsf{Split}\ n\ id\ a'_1\ \ldots\ a'_m\\
\vdash\ (\texttt{PropSplitUp}\ id\ a,\ S,\ G\left[\begin{array}{l} a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_m \\ a_i \xrightarrow{G} \mathsf{Embed}{-}\tau\ a'_i, 1 \leq i \leq m \end{array}\right],\ D)
\end{array}
$$

As with Pair nodes, the rules for other nodes with exactly one child are like those for Embed$-\tau$, and are listed in Appendix A.

Evaluating RjnPtr Nodes

As the graph model suggests, it is not necessary to unshare nodes at the top of an area of quasishared graph based solely on the nodes' evaluation. Subsequent evaluation of a Split node will cause unsharing if necessary. Evaluation of a Rejoin node to the extent required by a certain command should then cause the same degree of evaluation to be performed to the quasishared expression. Since Rejoin nodes and quasishared graph areas are referenced indirectly, the machine will look for RjnPtr nodes instead of Rejoin nodes to be addressed from the stack.

$$\begin{aligned}
&(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a\\
&\vdash\ (\texttt{Eval-wh}; \texttt{Pop 1}; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Pop 1}; \texttt{Return}\rangle{:}\langle a{:}S, \texttt{Eval-wh}; C\rangle{:}D)\\
&(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a\\
&\vdash\ (\texttt{Eval-prw}; \texttt{Pop 1}; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Pop 1}; \texttt{Return}\rangle{:}\langle a{:}S, \texttt{Eval-prw}; C\rangle{:}D)
\end{aligned}$$

In particular, if the rejoined subexpression is in FNF, it will not contain a Split node, since the reduction to FNF would detect and propagate that node. Thus, that subexpression is in fact fully shared, rather than simply quasishared, so the Rejoin node may be removed.

$$\begin{aligned}
&(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a\\
&\vdash\ (\texttt{Eval-f}; \texttt{Pop 1}; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Remove}\rangle{:}\langle a{:}S, \texttt{Eval-f}; C\rangle{:}D)
\end{aligned}$$

where the `Remove` command is defined by

$$\begin{aligned}
&(\texttt{Remove}; C_0,\ a{:}[\,],\ G,\ \langle S, c; C\rangle{:}D),\ a \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ a'\ a_1\ \ldots\ a_N,\\
&\quad a_i \xrightarrow{G} \mathsf{RjnPtr}\ a\\
&\vdash\ (C,\ S,\ G\left[\begin{array}{c} a_i \mapsto \nabla\ a' \\ a \mapsto \nabla\ a' \end{array}\right],\ D)
\end{aligned}$$

One property implied by the above two rules will be useful later, and is explicitly stated here:

> Reduction to full normal form will remove all RjnPtr, Rejoin and Split nodes from an expression.

One consequence of this treatment of RjnPtr nodes is that commands may have to "look under" the Rejoin node for their command's argument.

Deconstructing and Unwinding RjnPtr Nodes

A Rejoin node must be propagated downward when references to the subexpressions of the shared node are made, so that these subexpressions may themselves be quasishared. For example, if the left element of a quasishared graph denoting a pair is to be returned, then the delimiting Rejoin node should be moved downward, from above the Pair node[3] to below it. Moreover, we must generalize this case to finding any number of nested Rejoin nodes enclosing the Pair node:

[3] The `Left` command expects the top element of the stack to be in WHNF. Since the execution of `Eval` would be passed on below the Rejoin node, it is safe to expect that the node below the Rejoin node should be a Pair node.

$$
\begin{array}{l}
(\mathtt{Left}; C,\, a_n^i\!:\!S,\, G,\, D),\;\; i \in \{1,\, \ldots,\, N_m\},\\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\; e_m,\;\; 1 \le m \le n,\;\; 1 \le j \le N_m,\\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\; N_m\; id_m\; a_{m-1}^k\; a_m^1\; \ldots\; a_m^{N_m},\;\; 2 \le m \le n,\text{ some } k,\, 1 \le k \le N_{m-1},\\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\; N_1\; id_1\; e_0\; a_1^1\; \ldots\; a_1^{N_1}\\
\qquad e_0 \xrightarrow{G} \mathsf{Pair}\; r_1\; r_2\\
\vdash\;\; (\mathtt{Left}; C,\, a_n^i\!:\!S,\, G \left[\begin{array}{l} \left. \begin{array}{l} a_1^j \mapsto \mathsf{Pair}\; b_j\; d_j \\ b_j \mapsto \mathsf{RjnPtr}\; b \\ d_j \mapsto \mathsf{RjnPtr}\; d \end{array} \right\} 1 \le j \le N_1 \\ b \mapsto \mathsf{Rejoin}\; N_1\; id_1\; r_1\; b_1\; \ldots\; b_{N_1} \\ d \mapsto \mathsf{Rejoin}\; N_1\; id_1\; r_1\; d_1\; \ldots\; d_{N_1} \end{array} \right],\, D)
\end{array}
$$

We can generalize this rule to arbitrary commands as follows:

$$
\begin{array}{l}
(c; C,\, a_n^i\!:\!S,\, G,\, D),\;\; i \in \{1,\, \ldots,\, N_m\},\\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\; e_m,\;\; 1 \le m \le n,\;\; 1 \le j \le N_m,\\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\; N_m\; id_m\; a_{m-1}^k\; a_m^1\; \ldots\; a_m^{N_m},\;\; 2 \le m \le n,\text{ some } k,\, 1 \le k \le N_{m-1},\\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\; N_1\; id_1\; e_0\; a_1^1\; \ldots\; a_1^{N_1}\\
\qquad e_0 \xrightarrow{G} S\; u_1\; \ldots\; u_\phi,\;\; S \ne \mathsf{RjnPtr}\\
\vdash\;\; (c; C,\, a_n^i\!:\!S,\, G \left[\begin{array}{l} a_1^j \mapsto S\; r'_{1,j}\; \ldots\; r'_{\phi,j},\;\; 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\; r_i,\;\; 1 \le i \le \phi,\;\; 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\; N_1\; id_1\; u_i\; r'_{i,1}\; \ldots\; r'_{i,N_1} \end{array} \right],\, D)
\end{array}
$$

The generalized rule applies to `Right` and other deconstruction commands, and as well for `Unwind` and other commands which must access graph elements as part of regular reduction. In addition, where the quasishared node is trivially fully shared because it is (say) These clauses are listed with each relevant command in Appendix A.

Evaluating Diseq Nodes

Expressions headed by Diseq nodes, like those headed by Scoped-S nodes, are in WHNF:

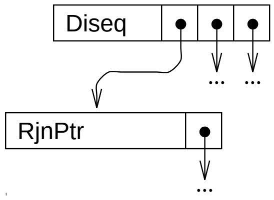
$$(\mathtt{Eval\text{-}wh}; C,\, a\!:\!S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Diseq}\; e_1\; e_2\; a' \;\vdash\; (C,\, a\!:\!S,\, G,\, D).$$

Evaluating Diseq nodes to PRWNF or FNF involves several issues:

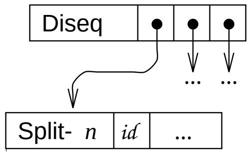
1. The expressions representing the values to be considered unequal may be altered through narrowing, and such changes must be managed.
2. The assertion of inequality between an atom and a quantifier ranging over atoms must be moved from Diseq nodes in the spine to an Ineq node.
3. Diseq nodes asserting the inequality of equal values reduce to the empty set; of trivially unequal expressions, to an indirection.
4. To make sure that no occurance of narrowing alters a Diseq node, its child subexpression must be reduced to FNF.
5. The derivation rules of §4.5.3 expect to find disequality nodes at the base of stacks stored in the dump.

A potential narrowing of quantifiers mentioned in Diseq nodes may be manifested in several ways:

- by the presence of RjnPtr nodes as or in those subexpressions,



- by the presence of a Split node as or in one of the subexpressions,



- by the presence of anything as the immediate child of a Rejoin node besides a graph node which is itself in full normal form.

In fact, the last item subsumes the first two: since Diseq nodes are created with logic variables as the child expressions asserted to be unequal, a discovery of anything else as those children is evidence that narrowing may have affected that child. Moreover, we can use here the observation of p. 77 that reduction to FNF removes RjnPtr nodes and other quasisharing delimiters. Thus, encountering a Diseq node whose disequaled children are not LVar nodes should first reduce those nodes to FNF:

$$
\begin{aligned}
&(\texttt{Eval-}\mathit{form}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a' \\
&\quad \vdash\ (\texttt{Eval-f}; \texttt{Pop}\ 1; \texttt{Eval-f}; \ldots,\, e_1{:}e_2{:}\ldots,\ \ldots,\ \ldots)
\end{aligned}
$$

Clearly, this derivation rule is not in its final form; we shall return to this part of the derivation later.

Once the disequaled expressions are in FNF, they can be examined to eliminate redundant structures, according to rules taken from some basic inequality identities:

$$
\begin{aligned}
(e_1)_\tau \neq (e_2)_\tau &\iff e_1 \neq e_2 \\
\tau_1 \neq \tau_2 &\implies (e_1)_\tau \neq (e_2)_\tau \\
\langle e_1, e_2\rangle \neq \langle e_3, e_4\rangle &\iff e_1 \neq e_3\ \vee\ e_2 \neq e_4
\end{aligned}
$$

Also at this point, the conversion to Ineq nodes (as specified by item (2) above) and immediate reductions (implied by item (3)) can be performed. A new command is introduced here to perform the reductions, `DiseqPreExamine`:

$$
\begin{aligned}
&(\texttt{DiseqPreExamine}; C,\, a{:}[\,],\, G,\, D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
&\qquad e_1 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e_1',\ e_2 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e_2' \\
&\quad \vdash\ (\texttt{DiseqPreExamine}; C,\, a{:}[\,],\, G[a \mapsto \mathsf{Diseq}\ e_1'\ e_2'\ a'],\, D) \\
&(\texttt{DiseqPreExamine}; C_0,\, a{:}[\,],\, G,\, \langle S, C\rangle{:}D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
&\qquad e_1 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ e_1',\ e_2 \xrightarrow{G} \mathsf{Embed}{-}\tau_2\ e_2',\ \tau_1 \neq \tau_2 \\
&\quad \vdash\ (C,\, a'{:}S,\, G[a \mapsto \nabla\ a'],\, D)
\end{aligned}
$$

$$\begin{array}{l}
(\texttt{DiseqPreExamine}; C_0,\ a{:}[\,],\ G,\ \langle S, C\rangle{:}\,D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
\qquad e_1 \xrightarrow{G} \mathsf{Pair}\ p_1\ p_3,\ \ e_2 \xrightarrow{G} \mathsf{Pair}\ p_2\ p_4 \\
\ \vdash\ \ (\texttt{Unwind}; \texttt{Retry},\ a{:}[\,],\ G \left[\begin{array}{ll}
t'_1 \mapsto \mathsf{Ap}\ \gamma_e\ e_1 & t''_1 \mapsto \mathsf{Ap}\ \gamma_e\ e_2 \\
t'_2 \mapsto \mathsf{Ap}\ t'_1\ e_3 & t''_2 \mapsto \mathsf{Ap}\ t''_1\ e_4 \\
n_2 \mapsto \mathsf{Ap}\ n_1\ t'_2 & n_2 \mapsto \mathsf{Ap}\ \gamma_a\ t''_2 \\
a \mapsto \mathsf{Ap}\ a_2\ a' & a_2 \mapsto \mathsf{Ap}\ a_1\ \gamma_\phi \\
a_1 \mapsto \mathsf{Ap}\ \gamma_i\ n_2 & \gamma_\phi \mapsto \mathsf{Phi} \\
\gamma_a \mapsto \mathsf{Fun}\ 2\ \texttt{_and} & \gamma_i \mapsto \mathsf{Fun}\ 3\ \texttt{_if} \\
\gamma_e \mapsto \mathsf{Fun}\ 2\ \texttt{_equals} &
\end{array}\right],\ \langle S, C\rangle{:}\,D) \\
(\texttt{DiseqPreExamine}; C_0,\ a{:}[\,],\ G,\ \langle S, c; C\rangle{:}\,D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
\qquad e_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ e_2 \xrightarrow{G} \mathsf{Atom}\ \alpha,\ \ \ell \xrightarrow{G} \mathsf{Scope{-}Atoms}\ e\ I \\
\ \vdash\ \ (C,\ a'{:}S,\ G \left[\begin{array}{l}
a \mapsto \nabla\ a' \\
\ell \mapsto \mathsf{Scope{-}Atoms}\ e\ I' \\
I' \mapsto \mathsf{Ineq}\ I\ \alpha
\end{array}\right],\ D) \\
(\texttt{DiseqPreExamine}; C_0,\ a{:}[\,],\ G,\ \langle S, c; C\rangle{:}\,D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
\qquad e_1 \xrightarrow{G} N,\ \ e_2 \xrightarrow{G} N,\ \ N \in \{\mathsf{Atom}\ \alpha,\ \ \mathsf{Bool}\ \beta,\ \ \mathsf{Logic\ Var}\ \ell\} \\
\ \vdash\ \ (C,\ a{:}S,\ G\,[a \mapsto \mathsf{Phi}]\,,\ D) \\
(\texttt{DiseqPreExamine}; C_0,\ a{:}[\,],\ G,\ \langle S, C\rangle{:}\,D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a', \\
\qquad \text{and} \left\{\begin{array}{c}
\textit{-either-} \\
e_1 \xrightarrow{G} \mathsf{Atom}\ \alpha_1,\ \ e_2 \xrightarrow{G} \mathsf{Atom}\ \alpha_2,\ \ \alpha_1 \neq \alpha_2 \\
\textit{-or-} \\
e_1 \xrightarrow{G} \mathsf{Bool}\ \beta_1,\ \ e_2 \xrightarrow{G} \mathsf{Bool}\ \beta_2,\ \ \beta_1 \neq \beta_2
\end{array}\right\} \\
\ \vdash\ \ (C,\ a'{:}S,\ G\,[a \mapsto \nabla\ a']\,,\ D) \\
\text{otherwise},\ \ (\texttt{DiseqPreExamine}; C,\ S,\ G,\ D) \vdash (C,\ S,\ G,\ D)
\end{array}$$

Finally, the ideas and commands above can be assembled together:

$$\begin{array}{l}
(E; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',\ \ E \in \{\texttt{Eval-prw},\ \texttt{Eval-f}\} \\
\quad \vdash\ \ (\texttt{_eval_diseq},\ e_1{:}\,e_2{:}\,a{:}[\,],\ G,\ \langle a{:}S, E; C\rangle{:}\,D)
\end{array}$$

where the code sequence `_eval_diseq` is given in Figure 4.10. This mechanism does preserve Diseq nodes on the stack in the way required by the process of §4.5.3, p. 70.

Substitution of Logic Variables

The introduction of LVUpdate nodes and their interactions with the standard commands (described in the graph model of §3.3.2 and introduced into the NG-machine in §4.4.2) are readily described. The `MkLVarSwap` command is defined as

$$(\texttt{MkLVarSwap}; C,\ e{:}\,q{:}\,\ell{:}\,S,\ G,\ D)\ \ \vdash\ \ (C,\ S,\ G[a \mapsto \mathsf{LVUpdate}\ q\ \ell\ e],\ D).$$

The derivation rule for the evaluation of an LVUpdate node is

$$\begin{array}{l}
(c; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a' \\
\quad \vdash\ \ (\texttt{PerformUpdate},\ a{:}[\,],\ G,\ \langle a{:}S, c; C\rangle{:}\,D),
\end{array}$$

where $c \in \{\texttt{Eval-wh},\ \texttt{Eval-prw},\ \texttt{Eval-f}\}$, sets up the actual substitution, which is defined as follows:

```
_eval_diseq:  Eval-f
              Pop 1
              Eval-f
              Pop 1
              DiseqPreExamine
              Push 0
              GetDisequaledSubexpr
              Eval-wh
              SetCase _done _force _union _force
      _done:  Pop 2
              Return
     _force:  Eval-f
              SetCase _done _done _union _done
     _union:  UnionElements
              Push 1
              Push 3
              GetTwoDisequals
              MkDisequal
              Push 1
              Push 4
              GetTwoDisequals
              MkDisequal
              MkUnion
              Update 3
              Pop 2
              Retry
```

Figure 4.10: The code sequence `_eval_diseq`.

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} \nabla\ e$$
$$\vdash\ (\texttt{PerformUpdate},\ a\!:\![\,],G\,[a \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ e]\,,D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ \langle S,C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} \mathsf{Logic\ Var}\ \alpha$$
$$\vdash\ (C,\ S,\ G\,[a \mapsto \nabla\ \beta]\,,\ D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ \langle S,C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} N,$$
$$N \in \ \{\mathsf{Phi},\ \mathsf{Atom}\ i,\ \mathsf{Bool}\ b,\ \mathsf{Logic\ Var}\ \alpha',\ \ \alpha \neq \alpha'\}$$
$$\vdash\ (C,\ S,\ G\,[a \mapsto \nabla\ a']\,,\ D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ \langle S,C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} N\ a_1$$
$$N \text{ the tag of a node with one child in the graph, but not an indirection node}$$
$$\vdash\ (C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \end{bmatrix},\ D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ \langle S,C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} N\ a_1\ a_2$$
$$N \text{ the tag of a node with two children in the graph}$$
$$\vdash\ (C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1'\ a_2' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \\ a_2' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_2 \end{bmatrix},\ D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ \langle S,C\rangle\!:D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} N\ a_1\ a_2\ a_3$$
$$N \text{ the tag of a node with three children in the graph}$$
$$\vdash\ (C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1'\ a_2'\ a_3' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \\ a_2' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_2 \\ a_3' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_3 \end{bmatrix},\ D)$$

$$(\texttt{PerformUpdate},\ a\!:\![\,],G,\ D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ \ a' \xrightarrow{G} \mathsf{LVUpdate}\ \alpha'\ \beta'\ a''$$
$$\vdash\ (\texttt{PerformUpdate},\ a'\!:\![\,],\ G,\ \langle a\!:\![\,],\texttt{PerformUpdate}\rangle\!:D)$$

4.6 Self-Reference

To correctly implement sets which contain references to themselves, a small amount of additional work is required. To avoid incorrect graph cycles, a modified access to global supercombinators is needed: for non-zero arity functions the result is the same as for the `Push Global` command, but for zero-arity functions we specify a new command which duplicates the reference:

$$(\texttt{Push SafeGlobal}\ i; C,\ S,\ G,\ D),\ \gamma_i \xrightarrow{G} \mathsf{Fun}\ 0\ c\ \vdash\ (C,\ a\!:\!S,\ G[a \mapsto \mathsf{Fun}\ 0\ c],\ D),\ \gamma_i \neq a$$
$$(\texttt{Push SafeGlobal}\ i; C,\ S,\ G,\ D),\ \gamma_i \text{ otherwise} \vdash\ (C,\ \gamma_i\!:\!S,\ G,\ D)$$

In conjuction with the naming convention of §2.2.2 for supercombinators whose bodies are quatifiers, we shall use this command in the translation of the use of marked references in §4.8.4.

4.7 Extending the `Print` Command

The `Print` command for the NG-machine includes the rules for non-set-based constructs for the G-machine, and is extended to output set expressions as follows:

```
_print_singleton: PrintMesg {
                  Eval-wh
                  Eval-prw
                  Print
                  PrintMesg }
                  Return
```

Figure 4.11: The code sequence `_print_singleton`.

```
_print_union Print
             Query
             Jfalse _exit
             Eval-wh
             Eval-prw
             Print
             Return
       _exit Pop 1
             Return
```

Figure 4.12: The code sequence `_print_union`.

- The representation of the empty set is simply added to the output stream:

$$(O, \texttt{Print}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Phi}\ \vdash\ (O; \phi,\ C,\ S,\ G,\ D)$$

- Singleton sets are printed in between curly braces:

$$(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Singleton}\ a'\ \vdash\ (\texttt{\_print\_singleton},\ a'{:}[\,],\ G,\ \langle S, C\rangle{:}D)$$

  where the code sequence `_print_singleton` is defined as shown in Figure 4.11, and the command `PrintMesg` is defined by

$$(O, \texttt{PrintMesg}\ \ c; C,\ S,\ G,\ D)\ \vdash\ (O; c,\ C,\ S,\ G,\ D)$$

- Issues involved in printing binary unions of sets were discussed in §4.3. After printing one "side" of the union, the user should be prompted to determine whether additional members of the set should be generated. The `Query` command will be used without a formal definition, but the behavior of the command is as follows: the user will be prompted, and the reply will be reflected in a new element placed on the top of the stack. A boolean value of **true** indicates that additional elements *should* be generated; a value of *false* indicates that additional elements *should not* be generated. Then the definition of the `Print` command for binary union is

$$(\texttt{Print}; C,\ a{:}S,\ G,\ D)\ \vdash\ (\texttt{\_print\_union},\ a_1{:}\,a_2{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}D)$$

  where the code sequence `_print_union` is defined as shown in Figure 4.12.

- Printing a set union over a logic variable is similar to printing tagged values. The function γ will be assumed to put the address of a scope declaration node into some printable from, *i.e.*, an alphanumeric string; then the `PrintAddr` command is defined as follows:

```
_print_lv_abstr: Push 0
                 GetScopedValue
                 Push 1
                 Push 0
                 PrintMesg U_[
                 PrintAddr
                 PrintMesg ∈
                 PrintRange
                 PrintMesg ]
                 Print
                 Pop 1
                 Return
```

Figure 4.13: The code sequence `_print_lv_abstr`.

$$(O,\, \texttt{PrintAddr}; C,\, a{:}S,\, G,\, D) \;\vdash\; (O; \gamma(a),\, C,\, S,\, G,\, D)$$

and the `PrintRange` command as:

$$\begin{array}{l}
(O,\, \texttt{PrintRange}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Scope{-}Hu}\; a'\; I \;\vdash\; (O; \texttt{hu},\, C,\, S,\, G,\, D)\\
(O,\, \texttt{PrintRange}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Scope{-}Atoms}\; a'\; I_0,\; I_0 \xrightarrow{G} \mathsf{Ineq}\; I_1\; a_1,\; \ldots\\
\qquad I_i \xrightarrow{G} \mathsf{Ineq}\; I_{i+1}\; a_{i+1},\; \ldots I_n \xrightarrow{G} \mathsf{Ineqs{-}End}\\
\;\vdash\; (O; \texttt{atoms} \setminus \{A_{a_1},\; \ldots,\; A_{a_n}\},\, C,\, S,\, G,\, D)\\
(O,\, \texttt{PrintRange}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Scope{-}Bools}\; a'\; I \;\vdash\; (O; \texttt{bools},\, C,\, S,\, G,\, D)
\end{array}$$

The `print` command can then be extended,

$$\begin{array}{l}
(\texttt{Print}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Scope{-}}S\; a'\; I\\
\;\vdash\; (\texttt{_print_lv_abstr},\, a{:}[\,],\, G,\, \langle S, \texttt{Print}; C\rangle{:}\, D)
\end{array}$$

where the code sequence `_print_lv_abstr` is defined as shown in Figure 4.13. Since the subexpression of a PRWNF expression headed by a scoping delimiter is in FNF, no evaluation of that subexpression must be done before printing it.

- The function γ is also used for displaying logic variables:

$$(O,\, \texttt{Print}; C,\, a{:}S,\, G,\, D)\, a \xrightarrow{G} \mathsf{Logic\ Var}\; \ell \;\vdash\; (O; \gamma(\ell),\, C,\, S,\, G,\, D)$$

- Printing disequality nodes is similar to the scoping delimiters. The derivation rule is:

$$\begin{array}{l}
(\texttt{Print}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{Diseq}\; e_1\; e_2\; a'\\
\;\vdash\; (\texttt{_print_diseq},\, e_1{:}e_2{:}a'{:}[\,],\, G,\, \langle S, \texttt{Print}; C\rangle{:}\, D)
\end{array}$$

and the code sequence `_print_diseq` as shown in Figure 4.14.

- Since the no narrowing will occur at the top level of an expression in PRWNF, encountering a Rejoin node is not a problem since the graph it delimits is thus fully shared:

$$\begin{array}{l}
(\texttt{Print}; C,\, a{:}S,\, G,\, D),\;\; a \xrightarrow{G} \mathsf{RjnPtr}\; a',\;\; a' \xrightarrow{G} \mathsf{Rejoin}\; n\; id\; e\; \ldots\\
\;\vdash\; (\texttt{Print}; C,\, e{:}S,\, G,\, D)
\end{array}$$

```
_print_diseq: PrintMesg _[
              Print
              PrintMesg ≠
              Print
              PrintMesg ]
              Print
```

Figure 4.14: The code sequence _print_diseq.

4.8 Translating PowerFuL Programs to NG Code

Translating programs from the syntax of PowerFuL into NG-code requires a few simple extensions to the G-code translation schema, based on the extensions to the G-machine. First, the use of multistage evaluation must be reflected by the top-level rule for supercombinators (§4.8). The basic set-denoting expressions are translated in a way similar to ordinary constructors (§4.8.1). The translation of identifiers ranging over sets is described in §4.8.2. Finally, the translation schemes are extended to marked supercombinators in §4.8.3.

4.8.1 Multistage Evaluation

Since it is expressions in PRWNF, and not WHNF, which are suitable for printing, the translation scheme should produce NG-code which will reduce the graph to that form. The $\mathcal{P}ROG$ scheme is the relevant translation rule:

$$\begin{aligned}
&\mathcal{P}ROG[\![\$\mathbf{S}_0\ x_{00}\ \ldots x_{0r_0} = E_0; \ldots; \$\mathbf{S}_n\ x_{n0}\ \ldots x_{nr_n} = E_n]\!] \\
&= \ \texttt{Alloc Globals}\ (n+1); \texttt{Set Global}\ 0\ r_0\ \Gamma_0; \ldots; \texttt{Set Global}\ n\ r_n\ \Gamma_n; \\
&\quad \texttt{Push Global}\ 0; \texttt{Eval-wh}; \texttt{Eval-prw}; \texttt{Print}; \texttt{Halt}; \\
&\quad \Gamma_0 : \mathcal{SC}[\![\$\mathbf{S}_n\ x_{00}\ \ldots x_{0r_0} = E_0]\!]; \\
&\quad \vdots \\
&\quad \Gamma_n : \mathcal{SC}[\![\$\mathbf{S}_n\ x_{n0}\ \ldots x_{nr_n} = E_n]\!];
\end{aligned}$$

4.8.2 Set Constructs

Translating basic set constructs is fairly straightforward. To translate the empty set, we simply generate code to push the Phi node onto the stack:

$$\mathcal{C}[\![\phi]\!]\,\rho\,d = \texttt{Push Phi}.$$

To translate the singleton set we construct an instance of its element, and then place the element under a Singleton node:

$$\mathcal{C}[\![\{x\}]\!]\,\rho\,d = \mathcal{C}[\![x]\!]\,\rho\,d; \texttt{MkSingleton}.$$

To make a graph corresponding to the union of two expressions, each of the two are constructed and then joined by a Union node:

$$\mathcal{C}[\![x_1 \cup x_2]\!]\,\rho\,d = \mathcal{C}[\![x_2]\!]\,\rho\,d; \mathcal{C}[\![x_1]\!]\,\rho\,(d+1); \texttt{MkUnion}$$

Finally, to translate the distribution of a function over a set, the following rule applies:

$$\mathcal{C}[\![f^+ S]\!]\,\rho\,d = \mathcal{C}[\![S]\!]\,\rho\,d; \mathcal{C}[\![f]\!]\,\rho\,(d+1); \texttt{Push Global \$Over}; \texttt{MkAp}\ 2$$

where the \$Over global refers to the code sequence of Figure 4.15.

```
 _Over:   Push 1
          Eval
          SetCase _Over_1 _Over_2 _Over_3 _Over_4
_Over_1:  Update 3
          Pop 2
          Unwind
_Over_2:  Push 1
          MkAp 1
          Update 3
          Pop 2
          Unwind
_Over_3:  UnionElements
          Push 2
          Push Global $Over
          MkAp 2
          Push 2
          Push 2
          Push Global $Over
          MkAp 2
          MkUnion
          Update 3
          Pop 2
          Unwind
_Over_4:  Push 0
          GetScopedValue
          Push 1
          Push Global $Over
          MkAp 2
          CompleteLVScope
          Update 3
          Pop 2
          Unwind
```

Figure 4.15: The code sequence for the arity two operator to distribute a function over a set.

4.8.3 Logic Variables

Translating the declaration of a logic variable is somewhat more complicated, reflecting the combination of NG-code instructions needed to construct expressions over unbound values. To construct the graph of an expression of the form

$$\bigcup_{x \in \mathbf{hu}} E$$

the following steps must be taken:

1. Construct a declaration of scope for x, without indication of an expression over which x should range, and a reference to the scoped value.
2. Construct an instance of E, noting the new stack depth and the offset for x.
3. Update the scope declaration to point ot the graph of E.
4. Leave only the scope declaration on top of the previous stack.

Thus the translation scheme is:

$$\begin{aligned} \mathcal{C}[\![\bigcup_{x \in \mathbf{hu}} e]\!]\, \rho\, d &= \texttt{NewLVar}; \mathcal{C}[\![e]\!]\, \rho \left[\frac{d+2}{x} \right] (d+2); \\ & \quad \texttt{Slide } 1; \texttt{CompleteLVScope} \end{aligned}$$

4.8.4 Marked Supercombinators

To safely build the graph of sets which refer to themselves, we must translate marked supercombinators with the `Push SafeGlobal` command:

$$\mathcal{C}[\![\$!\mathrm{S}]\!]\, \rho\, d = \texttt{Push SafeGlobal } (\rho\ \$!\mathrm{S})$$

Unmarked supercombinators can be translated as before.

Chapter 5

Conclusion & Future Work

The NG-machine presented in this thesis represents a step in the evolution of functional-logic languages in that the design of this implementation is based more on functional language implementation techniques than on techniques for logic programming. This shift contrasts with previous work in this area, which took traditional techniques for logic programming languages as their starting point. The NG-machine retains both the conceptually simple structure of the G-machine and its inclusion of lazy, higher-order programming without restricting the use of logic variables generated in our source language. Other proposed implementations of functional-logic languages do not offer lazy evaluation semantics as part of the language [5, 6] or cannot elegantly incorporate higher-order values into the implementation [5, 3]. Moreover, because the NG-machine minus its mechanisms for manipulating sets is essentially a G-machine, the functional aspects of PowerFuL programs can be executed by the NG-machine as well as by a simple G-machine, which is generally considered to be a good implementation for lazy functional languages. Thus the NG-machine will provide an especially good implementation model for programs making frequent use of shared values and lazy evaluation, common programming techniques for lazy functional languages.

Future work on implementing PowerFuL and the NG-machine can proceed in several directions. Improvements can be made to the abstract machine, to the translation schema from PowerFuL to supercombinators and from supercombinators to NG-code, and to the PowerFuL language itself. Since this thesis has focused on the implementation of PowerFuL, we will not discuss the last of these directions, which we consider to be an issue of language design rather than language implementation. Beyond improvements and optimizations to the current architecture, the exploitation of parallel architectures will realize significant gains in performance.

The NG-machine itself would more efficiently use heap space by modifying and reusing the stacks encountered in propagation, rather than simply discarding them. Following propagation, a similarly structured sequence of stacks will be constructed. While the original sequence would reference either quasishared nodes or nodes forming an expression not yet narrowed, the new sequence will reference graph nodes created in the duplication of nodes that would have occurred when the quasisharing delimiters were propagated upward through the graph. By overwriting the sequence of stacks to refer to the appropriate new nodes as they are created, the time that would otherwise have been spent rebuilding these stacks will be saved. Such a modification is difficult in the current architecture because the reduction process does not notice to which of the n distinct trailing subgraphs a pointer to a quasishared area of graph refers. It may be possible to store this information in the dump, or associated with Rejoin nodes in the heap.

Optimizations can also be made to the translation schema. Several improvements to the original G-machine compilation schema have been presented [7, 8]. Unfortunately, many will not be directly applicable to the NG schema because they tend to make the G-machine more spineless. While more efficient for graph reduction, spinelessness is not compatible with the current NG-machine architecture. The intention of such optimizations is to generate G-code which will directly compute the graph of a weak head-normal form of an expression, rather than constructing the graph of the unreduced expression, and then reducing that graph. This sequence of reduction steps may initiate narrowing, but without the structure of the unreduced expression in the graph, our mechanism could not perform the required manipulations. An easy solution to this problem would be to construct the relevant graphs and tentatively ignore them, proceeding with the optimized behavior. If narrowing turns out to be unnecessary, then the unreduced graph can be reclaimed immediately. Otherwise, the graph will exist for the narrowing manipulations. While such a scheme will reap the benefits of the optimized behavior, time will still be spent — and perhaps wasted — constructing graphs which may never be used. A better but more ambitious solution would extend the narrowing mechanism to cope with some spinelessness.

Peephole optimizations to target code have been found to be an important phase of compilation in general, and Peyton Jones presented several peephole optimizations to G-code [7]. Excepting those dealing with `case` expressions, which the current definition of PowerFuL does not include, these optimizations are directly applicable to NG-code, and it would be worthwhile to identify other such improvements involving the new NG-code instructions.

Both the abstract machine and the compilation schema would benefit from applying program analysis and transformation techniques. Transformations such as those based on strictness and update analysis have provided substantial improvements to functional programs; similar improvements might also be gained from applying these analyses to PowerFuL. Further improvement should also result from new forms of analysis relevant to the set-based aspect of the language. In particular: discovering expressions known to be fully grounded allows quasisharing to be taken as full sharing. Discovering expressions in which narrowing will always take place will facilitate some compile-time reduction of manipulations which consume relatively large amounts of heap space at run-time. Finally, discovering expressions where, after narrowing into a union of sets, some of the expressions created in the narrowing reduce immediately to the empty set will allow a more efficient narrowing, where these expressions can simply not be created.

The style of pure, lazy functional programming has proven to be particularly well-suited for exploiting parallel architectures and phrasing parallel algorithms [7]. The G-machine in particular has been used as a basis for the compilation of parallel functional programming languages [8]. Given the amount of parallelism inherant to the semantics of PowerFuL [10] — especially the unions of sets generated in narrowing — the development of a parallel implementation of PowerFuL would provide a great improvement in performance and would be a highly significant work in itself. In general, algorithms for graph reduction machines do tend to be much more easily parallelizable than those for machines like the WAM, which are based on the inherently sequential backtracking algorithm.

Appendix A

Full Definition of the NG-Machine

An NG-machine is a state machine defined by a finite sequence of *NG-code instructions*, described below. The *state* of an NG-machine is given by a quintuple,

$$(O, C, S, G, D)$$

where

- O is a finite sequence of output symbols,
- C is a finite sequence of NG-code instructions,
- S is a stack of *addresses*,
- G is a graph modeled by a map from addresses to *graph nodes* and
- and D is the *dump*, a stack of pairs, each of which contains a finite sequence of code plus a stack of addresses.

Except when explicitly describing an output-related process, we will write the state as a quadruple, omitting the output. It is often necessary to specify the first NG-code instruction of an otherwise arbitrary sequence; a semicolon will separate particular instructions in an NG-code sequence written from left-to-right (the semicolon will not be used for instructions written in a vertical sequence). Similarly, a colon will separate the first element of the stack or dump from the rest of the sequence. The notation

$$a \xrightarrow{G} N$$

means that the graph G maps the address a to the node N; the notation

$$G[a \mapsto N]$$

describes a graph which maps a to N, and all other addresses to wherever G maps them. The *initial state* of an NG-machine given by the instruction sequence I is defined to be the tuple $([], I, [], \phi, [])$[1], where $[]$ is an empty sequence. An NG-machine state is said to be in a *final state* if it is of the form

$$(\texttt{Halt}; C, S, G, D)$$

that is, one having reached the NG-code instruction **`Halt`**.

Let K be the set of NG-machine states. The function $\vdash$, $\vdash: (K \rightarrow K)$, relates an NG-machine state with the state derivable from it, if one exists. We define this function below, in terms of the NG-code instruction at the beginning of the sequence in the first position of the state tuple, and in many cases also in terms of the other elements of the tuple.

[1] Or if excluding the output component, $(I, [], \phi, [])$.

A.1 The Halt Command

The **Halt** command[2] is trivially defined by its place in the definition of the final state of the abstract machine. Since it does halt the machine, it does not lead to any other states and thus does not appear on the left-hand side of any derivation clauses.

A.2 The Eval Family of Commands

The **Eval** commands reduce graph expressions to the indicated normal form.

A.2.1 The Eval-wh Command

The **Eval-wh** command is defined by the following clauses:

$$(\texttt{Eval-wh}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} N \ \vdash\ (C,\ n{:}S,\ G,\ D),$$
$$\text{where } N \in \{\ \mathsf{Phi},\ \mathsf{Singleton}\ a,\ \mathsf{Union}\ a_1\ a_2\ \mathsf{Logic\ Var}\ a,\ \mathsf{Scope}{-}S\ a\ I,\ \mathsf{Diseq}\ e_1\ e_2\ a,\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \mathsf{Fun}\ n\ i,\ n>0\ \}$$

$$(\texttt{Eval-wh}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ a_1\ a_2 \ \vdash\ (\texttt{Unwind},\ n{:}[\,],\ G,\ \langle S, \texttt{Eval-wh}; C\rangle{:}\,D)$$

$$(\texttt{Eval-wh}; C_0,\ n{:}S_0,\ G,\ D),\ n \xrightarrow{G} \mathsf{Fun}\ 0\ C \ \vdash\ (C,\ n{:}[\,],\ G,\ \langle S_0, \texttt{Eval-wh}; C_0\rangle{:}\,D)$$

$$(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \ \vdash\ (\texttt{Eval-wh}; C,\ a'{:}S,\ G,\ D)^3$$

$$(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n \ \vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G,\ D)^4$$

$$(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a$$
$$\vdash\ (\texttt{Eval-wh}; \texttt{Pop}\ 1; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Pop}\ 1; \texttt{Return}\rangle{:}\,\langle a{:}S, \texttt{Eval-wh}; C\rangle{:}\,D)^5$$

$$(\texttt{Eval-wh}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a'$$
$$\vdash\ (\texttt{PerformUpdate},\ a{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-wh}; C\rangle{:}\,D)^6$$

A.2.2 The Eval-prw Command

The **Eval-prw** command is defined by the following clauses:

[2] This command is discussed in §2.3.1, p. 12.
[3] This and the three preceeding derivation rules, §4.4.1, p. 57.
[4] This derivation rule, §4.5.4, p. 72.
[5] This derivation rule, §4.5.4, p. 77.
[6] This derivation rule, §4.5.4, p. 80.

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} N \ \vdash\ (C,\ a{:}S,\ G,\ D)$$
$$\text{where } N \in \{\ \mathsf{Phi},\ \mathsf{Singleton}\ a,\ \mathsf{Logic\ Var}\ a,\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Pair}\ a_1\ a_2\ \mathsf{Embed}{-}\tau\ a\ \mathsf{Fun}\ n\ i,\ n > 0\ \}$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \ \vdash\ (\texttt{Eval-prw}; C,\ a'{:}S,\ G,\ D)$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Union}\ a_1\ a_2 \ \vdash\ (\texttt{\_E2\_Union},\ a{:}[\,],\ G,\ \langle S, C\rangle{:}D)$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Scope}{-}S\ a'\ I \ \vdash\ (\texttt{\_f\_scoped},\ a'{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-prw}; C\rangle{:}D)$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\, a_1\ \ldots\ a_n \ \vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G,\ D)^{7}$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a$$
$$\vdash\ (\texttt{Eval-prw}; \texttt{Pop 1}; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Pop 1}; \texttt{Return}\rangle{:}\langle a{:}S, \texttt{Eval-prw}; C\rangle{:}D)^{8}$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a'$$
$$\vdash\ (\texttt{\_eval\_diseq},\ e_1{:}e_2{:}a{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-prw}; C\rangle{:}D)^{9}$$

$$(\texttt{Eval-prw}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a'$$
$$\vdash\ (\texttt{PerformUpdate},\ a{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-prw}; C\rangle{:}D)^{10}$$

with the auxiliary code sequences[11]

```
_E2_Union:  Push 0
            UnionElements
            Eval-wh
            Eval-prw
            SetCase _E2_Be _E2_Bss _E2_Bun _E2_Bss
_E2_Be:     Pop 1          _E2_Bun:  UnionElements
            Update 1                 Push 2
            Return                   Push 2
_E2_Bss:    Pop 2                    MkUnion
            Return                   Push 1
                                     MkUnion
                                     Update 4
                                     Pop 3
                                     Return
```

and

```
_f_scoped:  Push 0
            GetScopedValue
            Eval-wh
```

[6] This and the four preceeding derivation rules, §4.4.2, p. 57cmd:evalBbasicEnd.
[7] This derivation rule, §4.5.4, p. 72.
[8] This derivation rule, §4.5.4, p. 77.
[9] This derivation rule, §4.5.4, p. 79.
[10] This derivation rule, §4.5.4, p. 80.
[11] Given with the derivation rules which call them.

```
                SetCase _fs_e _fs_f _fs_bu _fs_f
_fs_bu: UnionElements           _fs_e:  Update 1
        NewLVar                         Return
        Push 4                  _fs_bu:  Eval-f
        Push 4                           SetCase _fs_e _fs_ok _fs_bu _fs_ok
        MkLVarSwap              _fs_ok:  Pop 1
        CompleteLVScope                  Return
        NewLVar
        Push 5
        Push 4
        MkLVarSwap
        CompleteLVScope
        MkUnion
        Update 3
        Pop 2
        Retry
```

and

```
_eval_diseq: Eval-f                                  _union: UnionElements
             Pop 1                                           Push 1
             Eval-f                                          Push 3
             Pop 1                                           GetTwoDisequals
             DiseqPreExamine                                 MkDisequal
             Push 0                                          Push 1
             GetDisequaledSubexpr                            Push 4
             Eval-wh                                         GetTwoDisequals
             SetCase _done _force _union _force              MkDisequal
       _done: Pop 2                                          MkUnion
             Return                                          Update 3
      _force: Eval-f                                         Pop 2
             SetCase _done _done _union _done                Retry
```

A.2.3 The Eval-f Command

The `Eval-f` command is defined by the following clauses:

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} N \ \vdash\ (C,\ a{:}S,\ G,\ D)$$
$$\text{where } N \in \{\ \mathsf{Phi},\ \mathsf{Logic\ Var}\ a,\ \mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Fun}\ n\ i,\ n > 0\ \}$$

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a' \ \vdash\ (\texttt{Eval-f}; C,\ a'{:}S,\ G,\ D)^{12}$$

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n \ \vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G,\ D)^{13}$$

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{RjnPtr}\ e,\ \ e \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e'\ a_1\ \ldots\ a_N,\ (\exists k)\, a_k = a$$
$$\vdash\ (\texttt{Eval-f}; \texttt{Pop 1}; \texttt{Return},\ e'{:}[\,],\ G,\ \langle e{:}[\,], \texttt{Fail}; \texttt{Remove}\rangle{:}\langle a{:}S, \texttt{Eval-f}; C\rangle{:}D)^{14}$$

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a'$$
$$\vdash\ (\texttt{\_eval\_diseq},\ e_1{:}e_2{:}a{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-f}; C\rangle{:}D)^{15}$$

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a'$$
$$\vdash\ (\texttt{PerformUpdate},\ a{:}[\,],\ G,\ \langle a{:}S, \texttt{Eval-f}; C\rangle{:}D)^{16}$$

and by the metarule

$$(\texttt{Eval-f}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathit{Node} \vdash (\mathit{CodeSequence},\ a{:}[\,],\ G,\ \langle S, \texttt{Eval-f}; C\rangle{:}\,D)^{17}$$

where the nodes and code sequences are according to Table 4.1, which is not reproduced in this Appendix to save space. The code sequence `_eval_diseq` is the same as that given in the previous section.

A.3 The `Unwind` Command

The `Unwind` command is defined by the following clauses:

$$(\texttt{Unwind}; C,\ n{:}S,\ G,\ D),\ n \xrightarrow{G} \mathsf{Ap}\ n'\ m\ \vdash\ (\texttt{Unwind}; C,\ n'{:}n{:}S,\ G,\ D)$$

$$\begin{aligned}(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_m{:}S,\ G,\ D),\ a_0 \xrightarrow{G} \mathsf{Fun}\ m\ C,\ a_i \xrightarrow{G} \mathsf{Ap}\ a_{i-1}\ n_i,\ 1 \le i \le m \\ \vdash\ (C,\ n_1{:}\ldots{:}n_m{:}a_m,\ G,\ \langle S, \texttt{Unwind}; \texttt{Unwind}\rangle{:}\,D)\end{aligned}$$

$$\begin{aligned}(\texttt{Unwind}; C_0,\ a_0{:}\ldots{:}a_k,\ G,\ \langle S, x; C\rangle{:}\,D),\ a_0 \xrightarrow{G} \mathsf{Fun}\ m\ C_1,\ k < m \\ \vdash\ (C,\ a_k{:}S,\ G,\ D)\end{aligned}$$

$$(\texttt{Unwind}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a'\ \vdash\ (\texttt{Unwind}; C,\ a'{:}S,\ G,\ D)$$

$$\begin{aligned}(\texttt{Unwind}; C_0,\ n{:}[\,],\ G,\ \langle S, x; C\rangle{:}\,D),\ n \xrightarrow{G} N\ \vdash\ & (C,\ n{:}S,\ G,\ D), \\ & \text{where } N \in \left\{ \begin{array}{l} \mathsf{Atom}\ \alpha, \\ \mathsf{Bool}\ \beta, \\ \mathsf{Pair}\ a_1\ a_2, \\ \mathsf{Embed}{-}\tau\ a, \\ \mathsf{Logic\ Var}\ \ell, \\ \text{set constructs} \end{array} \right\}^{18}\end{aligned}$$

$$\begin{aligned}(\texttt{Unwind}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_n \\ \vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G,\ D)^{19}\end{aligned}$$

$$\begin{aligned}&(\texttt{Unwind}; C,\ a_n^i{:}S,\ G,\ D),\ i \in \{1,\ \ldots,\ N_m\}, \\ &\quad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\ &\quad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\ &\quad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\ &\quad e_0 \xrightarrow{G} S\ u_1\ \ldots\ u_\phi,\ S \ne \mathsf{RjnPtr} \\ &\vdash\ (\texttt{Unwind}; C,\ a_n^i{:}S,\ G \left[\begin{array}{l} a_1^j \mapsto S\ r'_{1,j}\ \ldots\ r'_{\phi,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1} \end{array} \right],\ D)^{20}\end{aligned}$$

A.4 Control Commands

These commands — along with `Unwind` and the `Eval` commands — direct the flow of control of programs.

[12] This and the preceeding derivation rule, §4.4.3, p. 60.
[13] This derivation rule, §4.5.4, p. 72.
[14] This derivation rule, §4.5.4, p. 77.
[15] This derivation rule, §4.5.4, p. 79.
[16] This derivation rule, §4.5.4, p. 80.
[17] This rule, §4.4.3, p. 60.
[18] This and the four preceeding derivation rules, §4.1, p. 49.
[19] This derivation rule, §4.5.4, p. 72.
[20] This derivation rule, from the metarule of §4.5.4, p. 78.

A.4.1 The Return Command

The `Return` command[21] is defined by the following clauses:

$$(\texttt{Return}; C_0,\, [\,],\, G,\ \langle S, x; C\rangle{:}\, D) \ \vdash\ (C,\, S,\, G,\, D)$$
$$(\texttt{Return}; C_0,\, a{:}[\,],\, G,\ \langle S, x; C\rangle{:}\, D) \ \vdash\ (C,\, a{:}S,\, G,\, D)$$

A.4.2 The Jump Command

The `Jump` command[22] is defined by the following clause:

$$(\texttt{Jump}\ \ C; C_0,\, S,\, G,\, D) \vdash (C,\, S,\, G,\, D)$$

A.4.3 The Jfalse Command

The `Jfalse` command is defined by the following clauses:

$$(\texttt{Jfalse}\ C'; C,\, n{:}S,\, G,\, D),\ n \xrightarrow{G} \mathsf{Bool\ true}\ \vdash\ (C,\, S,\, G,\, D)$$
$$(\texttt{Jfalse}\ C'; C,\, n{:}S,\, G,\, D),\ n \xrightarrow{G} \mathsf{Bool\ false}\ \vdash\ (C',\, S,\, G,\, D)^{23}$$
$$(\texttt{Jfalse}\ C'; C,\, n{:}S,\, G,\, D),\ n \xrightarrow{G} \nabla\ n'\ \vdash\ (\texttt{Jfalse}\ C'; C,\, n'{:}S,\, G,\, D)^{24}$$
$$(\texttt{Jfalse}\ C'; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}S,\, G\left[\begin{array}{ll} \ell' \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell_2 \\ a_1 \mapsto \mathsf{Bool}\ \mathbf{true} & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Bool}\ \mathbf{false} & p_2 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \mathsf{UID} \end{array}\right],\, D)^{25}$$
$$(\texttt{Jfalse}\ C'; C,\, a_k{:}S,\, G,\, D),\ a_i \xrightarrow{G} \mathsf{RjnPtr}\ n,\ n \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ n'\ a_1\ \ldots\ a_N$$
$$\vdash\ (\texttt{Jfalse}\ C'; C,\, a_k{:}S,\, G\,[a_i \mapsto \nabla\ n']\,,\, D)^{26}$$

A.4.4 The SetCase Command

The `SetCase` command is defined by the following clauses:

$$(\texttt{SetCase}\ \ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Phi}$$
$$\vdash\ (L_1,\, a{:}S,\, G,\, D)$$
$$(\texttt{SetCase}\ \ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Singleton}\ a'$$
$$\vdash\ (L_2,\, a{:}S,\, G,\, D)$$
$$(\texttt{SetCase}\ \ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Union}\ a_1\ a_2$$
$$\vdash\ (L_3,\, a{:}S,\, G,\, D)$$
$$(\texttt{SetCase}\ \ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Scope{-}}S\ a'\ I$$
$$\vdash\ (L_4,\, a{:}S,\, G,\, D)$$
$$(\texttt{SetCase}\ \ L_1\ L_2\ L_3\ L_4; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a'$$
$$\vdash\ (L_4,\, a{:}S,\, G,\, D)$$

[21] This command, §4.1, p. 49.
[22] This command, §2.3.2, p. 16.
[23] This and the preceeding derivation rule, §2.3.3, p. 18.
[24] From the definition of an indirection node, §2.3.1, p. 13.
[25] This derivation rule, from the metarule of §4.5.3, p. 69.
[26] A consequence of the evaluation mechanism for RjnPtr nodes, §4.5.4, p. 77.

$$
\begin{array}{l}
(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C, a{:}S, G, D),\ a \xrightarrow{G} \nabla\ a' \\
\quad\vdash\ (\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C, a'{:}S, G, D)^{27} \\
(\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C, a_n^i{:}S, G, D),\ i \in \{1,\ \ldots,\ N_m\}, \\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\
\qquad e_0 \xrightarrow{G} S\ u_1\ \ldots\ u_\phi,\ S \ne \mathsf{RjnPtr} \\
\quad\vdash\ (\texttt{SetCase}\ L_1\ L_2\ L_3\ L_4; C, a_n^i{:}S, G \left[\begin{array}{l} a_1^j \mapsto S\ r'_{1,j}\ \ldots\ r'_{\phi,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1} \end{array}\right], D)^{28}
\end{array}
$$

A.4.5 The `Retry` Command

The **`Retry`** command[29] is defined by the following clauses:

$$(\texttt{Retry}; C_0, S_0, G, \langle S, C\rangle{:}D)\ \vdash\ (C, S, G, D)$$

A.5 The `Push` Family of Commands

There are a number of different commands in this collection, all of which add a pointer to the stack.

A.5.1 The `Push` Command

The **`Push`** command[30] is defined by the following clause:

$$(\texttt{Push}\ n; C, a_0{:}\ldots{:}a_n{:}S, G, D) \vdash (C, a_n{:}a_0{:}\ldots{:}a_n{:}S, G, D)$$

A.5.2 The `Push Atom` Command

The **`Push Atom`** command[31] is defined by the following clause:

$$(\texttt{Push Atom}\ \alpha; C, S, G, D) \vdash (C, n{:}S, G[n \mapsto \mathsf{Atom}\ \alpha], D)$$

A.5.3 The `Push Bool` Command

The **`Push Bool`** command[32] is defined by the following clause:

$$(\texttt{Push Bool}\ \beta; C, S, G, D) \vdash (C, n{:}S, G[n \mapsto \mathsf{Bool}\ \beta], D)$$

A.5.4 The `Push Global` Command

The **`Push Global`** command[33] is defined by the following clause:

$$(\texttt{Push Global}\ i; C, S, G, D) \vdash (C, \gamma_i{:}S, G, D)$$

[27] This and the five preceeding derivation rules, §4.2.4, p. 53.
[28] This derivation rule, from the metarule of §4.5.4, p. 78.
[29] This command, §4.4.2, p. 59.
[30] This command, §2.3.2, p. 16.
[31] This command, §2.3.2, p. 16.
[32] This command, §2.3.2, p. 16.
[33] This command, §2.3.2, p. 16.

A.5.5 The `Copy Global` Command

The `Copy Global` command[34] is defined by the following clause:

$$(\texttt{Push SafeGlobal } i; C,\, S,\, G,\, D),\ \gamma_i \xrightarrow{G} \mathsf{Fun}\ 0\ c \ \vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Fun}\ 0\ c],\, D),\ \gamma_i \neq a$$
$$(\texttt{Push SafeGlobal } i; C,\, S,\, G,\, D),\ \gamma_i \text{ otherwise} \ \vdash\ (C,\, \gamma_i\!:\!S,\, G,\, D)$$

A.5.6 The `Push Phi` Command

The `Push Phi` command[35] is defined by the following clause:

$$(\texttt{Push Phi}; C,\, S,\, G,\, D) \vdash (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Phi}],\, D)$$

A.6 Stack Manipulation

These commands manipulate pointers on the stack.

A.6.1 The `Pop` Command

The `Pop` command[36] is defined by the following clause:

$$(\texttt{Pop } n; C,\, a_1\!:\!\ldots\!:\!a_n\!:\!S,\, G,\, D) \vdash (C,\, S,\, G,\, D)$$

A.6.2 The `Slide` Command

The `Slide` command[37] is defined by the following clause:

$$(\texttt{Slide } n; C,\, a_0\!:\!\ldots\!:\!a_n\!:\!S,\, G,\, D) \vdash (C,\, a_0\!:\!S,\, G,\, D)$$

A.6.3 The `Update` Command

The `Update` command[38] is defined by the following clause:

$$(\texttt{Update } n; C,\, a_0\!:\!\ldots\!:\!a_n\!:\!S,\, G,\, D) \vdash (C,\, a_1\!:\!\ldots\!:\!a_n\!:\!S,\, G[a_n \mapsto \nabla\ a_0],\, D)$$

A.6.4 The `Alloc` Command

The `Alloc` command[39] is defined by the following clause:

$$(\texttt{Alloc } n; C,\, S,\, G,\, D) \vdash (C,\, a_1\!:\!\ldots\!:\!a_n\!:\!S,\, G[a_1 \mapsto \mathsf{Hole},\, \ldots,\, a_n \mapsto \mathsf{Hole}],\, D)$$

A.7 Construction Commands: The `Mk` Family

There are a number of different commands in this collection, all of which create a new graph node from the top one or two pointers on the stack, and replace those pointers with a new one.

A.7.1 The `MkAp` Command

The `MkAp` command[40] is defined by the following clause:

$$(\texttt{MkAp } n; C,\, a_0\!:\!\ldots\!:\!a_n\!:\!S,\, G,\, D) \vdash (C,\, e_n\!:\!S,\, G\left[\begin{array}{l} e_i \mapsto \mathsf{Ap}\ e_{i-1}\ a_i,\ i > 1 \\ e_1 \mapsto \mathsf{Ap}\ a_0\ a_1 \end{array}\right],\, D)$$

[34] This command, §4.6, p. 82.
[35] This command, §4.2.1, p. 50.
[36] This command, §2.3.2, p. 16.
[37] This command, §2.3.2, p. 16.
[38] This command, §2.3.2, p. 16.
[39] This command, §2.3.2, p. 16.
[40] This command, §2.3.2, p. 17.

A.7.2 The `MkPair` Command

The **`MkPair`** command[41] is defined by the following clause:

$$(\texttt{MkPair}; C,\, a_1\!:a_2\!:S,\, G,\, D) \;\vdash\; (C,\, a_*\!:S,\, G[a_* \mapsto \mathsf{Pair}\ a_1\ a_2],\, D)$$

A.7.3 The `MkEmbed` τ Commands

The **`MkEmbed`** τ commands[42] are defined by the following clause:

$$(\texttt{MkEmbed}\ \ \tau; C,\, a\!:S,\, G,\, D) \;\vdash\; (C,\, a_*\!:S,\, G[a_* \mapsto \mathsf{Embed}{-}\tau\ a],\, D)$$

A.7.4 The `MkSingleton` Commands

The **`MkSingleton`** commands[43] are defined by the following clause:

$$(\texttt{MkSingleton}; C,\, a\!:S,\, G,\, D) \vdash (C,\, a'\!:S,\, G[a' \mapsto \mathsf{Singleton}\ a],\, D)$$

A.7.5 The `MkUnion` Command

The **`MkUnion`** command[44] is defined by the following clause:

$$(\texttt{MkUnion}; C,\, a_1\!:a_2\!:S,\, G,\, D) \vdash (C,\, a'\!:S,\, G[a' \mapsto \mathsf{Union}\ a_1\ a_2],\, D),$$

A.7.6 Scoping Delimiters

Two commands are used to construct scoping delimiters.

The `NewLVar` Command

The **`NewLVar`** command[45] is defined by the following clause:

$$(\texttt{NewLVar}; C,\, S,\, G,\, D) \vdash (C,\, a_1\!:a_2\!:S,\, G\left[\begin{array}{l} a_1 \mapsto \mathsf{Logic\ Var}\ a_2 \\ a_2 \mapsto \mathsf{Scope{-}Hu}\ \bot\ a_3 \\ a_3 \mapsto \mathsf{Ineqs{-}End} \end{array}\right],\, D)$$

The `CompleteLVScope` Command

The **`CompleteLVScope`** command[46] is defined by the following clause:

$$\begin{array}{l} (\texttt{CompleteLVScope}; C,\, a_1\!:a_2\!:S,\, G,\, D),\ a_2 \xrightarrow{G} \mathsf{Scope{-}Hu}\ m\ n \\ \quad\vdash\ \ (C,\, a_2\!:S,\, G[a_2 \mapsto \mathsf{Scope{-}Hu}\ a_1\ n],\, D) \end{array}$$

A.7.7 The `MkDisequal` Command

The **`MkDisequal`** command[47] is defined by the following clause:

$$(\texttt{MkDisequal}; C,\, a_1\!:a_2\!:a'\!:S,\, G,\, D) \;\vdash\; (C,\, a\!:S,\, G[a \mapsto \mathsf{Diseq}\ a_1\ a_2\ a'],\, D).$$

[41] This command, §2.3.2, p. 17.
[42] This command, §2.3.2, p. 17.
[43] This command, §4.2.1, p. 50.
[44] This command, §4.2.1, p. 50.
[45] This command, §4.2.2, p. 51.
[46] This command, §4.2.2, p. 51.
[47] This command, §4.2.3, p. 53.

A.7.8 The MkLVarSwap Command

The `MkLVarSwap` command[48] is defined by the following clause:

$$(\texttt{MkLVarSwap}; C,\, e{:}\,q{:}\,\ell{:}\,S,\, G,\, D) \;\vdash\; (C,\, S,\, G[a \mapsto \mathsf{LVUpdate}\ q\ \ell\ e],\, D).$$

A.8 Deconstruction Commands

Just as the construction commands create new structures given children, these commands return the subgraphs of structures addresses at the top of the stack.

A.8.1 The Retract τ Commands

The `Retract` τ commands are defined by the following clauses:

$$\begin{array}{l}
(\texttt{Retract}\ \ \tau; C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a' \\
\ \vdash\ (C,\, a'{:}\,S,\, G,\, D)^{49} \\
(\texttt{Retract}\ \ \tau; C,\, a{:}\,S,\, G,\, D)\ \ \tau \in \{a,\ b,\ p\},\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I \\
\ \vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}\,S,\, G \left[\begin{array}{ll}
\ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' \\
a_1 \mapsto \mathsf{Embed{-}a}\ a_1' & \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 \\
a_1' \mapsto \mathsf{Logic\ Var}\ \ell_1 & \ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I \\
a_2 \mapsto \mathsf{Embed{-}b}\ a_2' & \ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I \\
a_2' \mapsto \mathsf{Logic\ Var}\ \ell_2 & \ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I \\
a_3 \mapsto \mathsf{Embed{-}p}\ a_3' & \ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I \\
a_3' \mapsto \mathsf{Pair}\ r_1\ r_2 & r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 \\
I \mapsto \mathsf{Ineqs{-}End} & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\
\multicolumn{2}{l}{p_1 \mapsto \mathsf{RjnPtr}\ e\quad p_2 \mapsto \mathsf{RjnPtr}\ e\quad p_3 \mapsto \mathsf{RjnPtr}\ e} \\
\multicolumn{2}{l}{e \mapsto \mathsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3} \\
\multicolumn{2}{l}{a \mapsto \mathsf{Split}\ 3\ id\ a_1\ a_2\ a_3 \qquad id \mapsto \mathsf{UID}}
\end{array}\right],\, D)^{50} \\
(\texttt{Retract}\ \ \tau; C,\, a{:}\,S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a' \\
\ \vdash\ (\texttt{Retract}\ \ \tau; C,\, a'{:}\,S,\, G,\, D) \\
(\texttt{Retract}\ \ \tau; C,\, a_n^i{:}\,S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\}, \\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\
\qquad e_0 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e' \\
\ \vdash\ (\texttt{Retract}\ \ \tau; C,\, a_n^i{:}\,S,\, G \left[\begin{array}{l}
a_1^j \mapsto \mathsf{Embed}{-}\tau\ r_j',\ 1 \le j \le N_1 \\
r_j' \mapsto \mathsf{RjnPtr}\ r,\ 1 \le j \le N_1 \\
r \mapsto \mathsf{Rejoin}\ N_1\ id_1\ e'\ r_1'\ \ldots\ r_{N_1}'
\end{array}\right],\, D) \\
(\texttt{Retract}\ \ \tau; C,\, a_n^i{:}\,S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\}, \\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\
\qquad e_0 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell \\
\ \vdash\ (\texttt{Retract}\ \ \tau; C,\, a_n^i{:}\,S,\, G\left[a_m^j \mapsto \nabla\ e_0,\ 1 \le m \le n,\ 1 \le j \le N_m\right],\, D)^{51}
\end{array}$$

[48] This command, §4.5.4, p. 80.

A.8.2 The `Left` Command

The `Left` command is defined by the following clauses:

$$\begin{array}{l}
(\texttt{Left}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 \\
\quad \vdash\ (C,\, a_1\!:\!S,\, G,\, D)^{52} \\
(\texttt{Left}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a' \\
\quad \vdash\ (\texttt{Left}; C,\, a'\!:\!S,\, G,\, D)^{53} \\
(\texttt{Left}; C,\, a_n^i\!:\!S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\}, \\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\
\qquad e_0 \xrightarrow{G} \mathsf{Pair}\ u_1\ u_2 \\
\quad \vdash\ (\texttt{Left}; C,\, a_n^i\!:\!S,\, G \left[\begin{array}{l} a_1^j \mapsto \mathsf{Pair}\ r'_{1,j}\ r'_{2,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le 2,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1},\ 1 \le i \le 2 \end{array}\right],\, D)^{54}
\end{array}$$

A.8.3 The `Right` Command

The `Right` command is defined by the following clauses:

$$\begin{array}{l}
(\texttt{Right}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 \\
\quad \vdash\ (C,\, a_2\!:\!S,\, G,\, D)^{55} \\
(\texttt{Right}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a' \\
\quad \vdash\ (\texttt{Right}; C,\, a'\!:\!S,\, G,\, D)^{56} \\
(\texttt{Right}; C,\, a_n^i\!:\!S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\}, \\
\qquad a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m, \\
\qquad e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1}, \\
\qquad e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1} \\
\qquad e_0 \xrightarrow{G} \mathsf{Pair}\ u_1\ u_2 \\
\quad \vdash\ (\texttt{Right}; C,\, a_n^i\!:\!S,\, G \left[\begin{array}{l} a_1^j \mapsto \mathsf{Pair}\ r'_{1,j}\ r'_{2,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le 2,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1},\ 1 \le i \le 2 \end{array}\right],\, D)^{57}
\end{array}$$

A.8.4 The `SingletonElement` Command

The `SingletonElement` command is defined by the following clause:

[49] This derivation rule, §2.3.3, p. 17.
[50] This derivation rule, §4.5.3, p. 69.
[51] This and the preceeding derivation rule, from the metarule of §4.5.4, p. 78.
[52] This derivation rule, §2.3.3, p. 18.
[53] From the definition of an indirection node, §2.3.1, p. 13.
[54] This derivation rule, from the metarule of §4.5.4, p. 78.
[55] This derivation rule, §2.3.3, p. 18.
[56] From the definition of an indirection node, §2.3.1, p. 13.
[57] This derivation rule, from the metarule of §4.5.4, p. 78.

$$(\texttt{SingletonElement}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Singleton}\ a'$$
$$\vdash\ (C,\, a'{:}S,\, G,\, D)$$[58]

$$(\texttt{SingletonElement}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{SingletonElement}; C,\, a'{:}S,\, G,\, D)$$

$$(\texttt{SingletonElement}; C,\, a_n^i{:}S,\, G,\, D),\ \ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \textsf{RjnPtr}\ e_m,\ \ 1 \le m \le n,\ \ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \textsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ \ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \textsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \textsf{Singleton}\ e'$$
$$\vdash\ (\texttt{SingletonElement}; C,\, a_n^i{:}S,\, G \left[\begin{array}{l} a_1^j \mapsto \textsf{Singleton}\ r'_j,\ 1 \le j \le N_1 \\ r'_j \mapsto \textsf{RjnPtr}\ r,\ 1 \le j \le N_1 \\ r \mapsto \textsf{Rejoin}\ N_1\ id_1\ e'\ r'_1\ \ldots\ r'_{N_1} \end{array}\right],\, D)$$[59]

A.8.5 The `UnionElements` Command

The `UnionElements` command is defined by the following clause:

$$(\texttt{UnionElements}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Union}\ a_1\ a_2$$
$$\vdash\ (C,\, a_1{:}a_2{:}S,\, G,\, D)$$[60]

$$(\texttt{UnionElements}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{UnionElements}; C,\, a'{:}S,\, G,\, D)$$[61]

$$(\texttt{UnionElements}; C,\, a_n^i{:}S,\, G,\, D),\ \ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \textsf{RjnPtr}\ e_m,\ \ 1 \le m \le n,\ \ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \textsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ \ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \textsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \textsf{Union}\ u_1\ u_2$$
$$\vdash\ (\texttt{UnionElements}; C,\, a_n^i{:}S,\, G \left[\begin{array}{l} a_1^j \mapsto \textsf{Union}\ r'_{1,j}\ r'_{2,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \textsf{RjnPtr}\ r_i,\ 1 \le i \le 2,\ 1 \le j \le N_1 \\ r_i \mapsto \textsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1},\ 1 \le i \le 2 \end{array}\right],\, D)$$[62]

A.8.6 The `GetScopedValue` Command

The `GetScopedValue` command is defined by the following clause:

$$(\texttt{GetScopedValue}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \textsf{Scope}{-}S\ a'\ I$$
$$\vdash\ (C,\, a'{:}S,\, G,\, D)$$[63]

$$(\texttt{GetScopedValue}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{GetScopedValue}; C,\, a'{:}S,\, G,\, D)$$[64]

---

[57] This derivation rule, §4.2.1, p. 50.
[58] From the definition of an indirection node, §2.3.1, p. 13.
[59] This derivation rule, from the metarule of §4.5.4, p. 78.
[60] This derivation rule, §4.2.1, p. 50.
[61] From the definition of an indirection node, §2.3.1, p. 13.
[62] This derivation rule, from the metarule of §4.5.4, p. 78.

$$(\texttt{GetScopedValue}; C,\, a_n^i\!:\!S,\, G,\, D),\ \ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \textsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \textsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some}\ k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \textsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \textsf{Scope}{-}S\ e'\ I$$
$$\vdash\ (\texttt{GetScopedValue}; C,\, a_n^i\!:\!S,\, G \begin{bmatrix} a_1^j \mapsto \textsf{Scope}{-}S\ r_j'\ I,\ 1 \le j \le N_1 \\ r_j' \mapsto \textsf{RjnPtr}\ r,\ 1 \le j \le N_1 \\ r \mapsto \textsf{Rejoin}\ N_1\ id_1\ e'\ r_1'\ \ldots\ r_{N_1}' \end{bmatrix},\, D)$$ [65]

A.8.7 The `GetDisequaledSubexpr` Command

The **`GetDisequaledSubexpr`** command is defined by the following clause:

$$(\texttt{GetDisequaledSubexpr}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \textsf{Diseq}\ a_1\ a_2\ a'$$
$$\vdash\ (C,\, a'\!:\!S,\, G,\, D)$$ [66]
$$(\texttt{GetDisequaledSubexpr}; C,\, a\!:\!S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{GetDisequaledSubexpr}; C,\, a'\!:\!S,\, G,\, D)$$ [67]
$$(\texttt{GetDisequaledSubexpr}; C,\, a_n^i\!:\!S,\, G,\, D),\ \ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \textsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \textsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some}\ k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \textsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \textsf{Diseq}\ A_1\ A_2\ e'$$
$$\vdash\ (\texttt{GetDisequaledSubexpr}; C,\, a_n^i\!:\!S,\, G \begin{bmatrix} a_1^j \mapsto \textsf{Diseq}\ A_1\ A_2\ r_j' I,\ 1 \le j \le N_1 \\ r_j' \mapsto \textsf{RjnPtr}\ r,\ 1 \le j \le N_1 \\ r \mapsto \textsf{Rejoin}\ N_1\ id_1\ e'\ r_1'\ \ldots\ r_{N_1}' \end{bmatrix},\, D)$$ [68]

A.8.8 The `GetTwoDisequals` Command

The **`GetTwoDisequals`** command is defined by the following clause:

$$(\texttt{GetTwoDisequals}; C,\, a\!:\!S,\, G,\, D),\ \ a \xrightarrow{G} \textsf{Diseq}\ a_1\ a_2\ a'$$
$$\vdash\ (C,\, a_1\!:\!a_2\!:\!S,\, G,\, D)$$ [69]
$$(\texttt{GetTwoDisequals}; C,\, a\!:\!S,\, G,\, D),\ \ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{GetTwoDisequals}; C,\, a'\!:\!S,\, G,\, D)$$ [70]

[63] This derivation rule, §4.2.2, p. 51.
[64] From the definition of an indirection node, §2.3.1, p. 13.
[65] This derivation rule, from the metarule of §4.5.4, p. 78.
[66] This derivation rule, §4.2.3, p. 53.
[67] From the definition of an indirection node, §2.3.1, p. 13.
[68] This derivation rule, from the metarule of §4.5.4, p. 78.

$$(\texttt{GetTwoDisequals}; C,\, a_n^i{:}S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \mathsf{Diseq}\ A_1\ A_2\ e'$$
$$\vdash\ (\texttt{GetTwoDisequals}; C,\, a_n^i{:}S,\, G \begin{bmatrix} a_1^j \mapsto \mathsf{Diseq}\ A_1\ A_2\ r_j' I,\ 1 \le j \le N_1 \\ r_j' \mapsto \mathsf{RjnPtr}\ r,\ 1 \le j \le N_1 \\ r \mapsto \mathsf{Rejoin}\ N_1\ id_1\ e'\ r_1'\ \ldots\ r_{N_1}' \end{bmatrix},\, D)^{71}$$

A.9 Condition Testing

These commands test expressions for certain properties.

A.9.1 The `Not` Command

The **`Not`** command is defined by the following clause:

$$(\texttt{Not}; C,\, n{:}S,\, G,\, D),\ n \xrightarrow{G} \mathsf{Bool}\ \beta$$
$$\vdash\ (C,\, n'{:}S,\, G[n' \mapsto \mathsf{Bool}\ \beta'],\, D),\ \beta' = {\sim}\beta \qquad ^{72}$$
$$(\texttt{Not}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{Not}; C,\, a'{:}S,\, G,\, D)^{73}$$
$$(\texttt{Not}; C,\, a{:}S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\, a{:}S,\, G \begin{bmatrix} \ell' \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell_2 \\ a_1 \mapsto \mathsf{Bool}\ \mathbf{true} & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Bool}\ \mathbf{false} & p_2 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \mathsf{UID} \end{bmatrix},\, D)^{74}$$
$$(\texttt{Not}; C,\, a_n^i{:}S,\, G,\, D),\ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} N,\ N \in \{\mathsf{Bool}\ \beta,\ \mathsf{Logic\ Var}\ \ell\}$$
$$\vdash\ (\texttt{Not}; C,\, a_n^i{:}S,\, G\left[a_m^j \mapsto \nabla\ e_0,\ 1 \le m \le n,\ 1 \le j \le N_m\right],\, D)^{75}$$

A.9.2 The `IsEmbedded` τ Commands

The **`IsEmbedded`** τ commands are defined by the following clauses:

[69] This derivation rule, §4.2.3, p. 53.
[70] From the definition of an indirection node, §2.3.1, p. 13.
[71] This derivation rule, from the metarule of §4.5.4, p. 78.
[72] This derivation rule, §2.3.3, p. 18.
[73] From the definition of an indirection node, §2.3.1, p. 13.
[74] This derivation rule, §4.5.3, p. 69.
[75] This derivation rule, from the metarule of §4.5.4, p. 78.

$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a'$$
$$\vdash\ (C,\ a'{:}S,\ G[a' \mapsto \mathsf{Bool\ true}],\ D)$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau'\ a',\ \tau \neq \tau'$$
$$\vdash\ (C,\ a'{:}S,\ G[a' \mapsto \mathsf{Bool\ false}],\ D)^{76}$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \nabla\ a'$$
$$\vdash\ (\texttt{IsEmbedded}\ \tau; C,\ a'{:}S,\ G,\ D)^{77}$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ \tau \in \{a,\ b,\ p\},\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}S,\ G \left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' \\ a_1 \mapsto \mathsf{Embed{-}a}\ a_1' & \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 \\ a_1' \mapsto \mathsf{Logic\ Var}\ \ell_1 & \ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I \\ a_2 \mapsto \mathsf{Embed{-}b}\ a_2' & \ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I \\ a_2' \mapsto \mathsf{Logic\ Var}\ \ell_2 & \ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I \\ a_3 \mapsto \mathsf{Embed{-}p}\ a_3' & \ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I \\ a_3' \mapsto \mathsf{Pair}\ r_1\ r_2 & r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 \\ I \mapsto \mathsf{Ineqs{-}End} & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\ \multicolumn{2}{l}{p_1 \mapsto \mathsf{RjnPtr}\ e\quad p_2 \mapsto \mathsf{RjnPtr}\ e\quad p_3 \mapsto \mathsf{RjnPtr}\ e} \\ \multicolumn{2}{l}{e \mapsto \mathsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3} \\ a \mapsto \mathsf{Split}\ 3\ id\ a_1\ a_2\ a_3 & \quad id \mapsto \mathsf{UID} \end{array}\right],\ D)^{78}$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a{:}S,\ G,\ D),\ \tau \in \{f,\ s\},\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$$
$$\vdash\ (C,\ a'{:}S,\ G\,[a' \mapsto \mathsf{Bool\ false}]\,,\ D)$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a_n^i{:}S,\ G,\ D),\ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some}\ k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e'$$
$$\vdash\ (\texttt{IsEmbedded}\ \tau; C,\ a_n^i{:}S,\ G \left[\begin{array}{l} a_1^j \mapsto \mathsf{Embed}{-}\tau\ r_j',\ 1 \le j \le N_1 \\ r_j' \mapsto \mathsf{RjnPtr}\ r,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r_1'\ \ldots\ r_{N_1}' \end{array}\right],\ D)$$
$$(\texttt{IsEmbedded}\ \tau; C,\ a_n^i{:}S,\ G,\ D),\ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some}\ k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} N,\ N \in \{\mathsf{Bool}\ \beta,\ \mathsf{Logic\ Var}\ \ell\}$$
$$\vdash\ (\texttt{IsEmbedded}\ \tau; C,\ a_n^i{:}S,\ G\,[a_m^j \mapsto \nabla\ e_0,\ 1 \le m \le n,\ 1 \le j \le N_m]\,,\ D)^{79}$$

A.9.3 The `Equals` Command

The `Equals` command is defined by the following clauses:

[76] This and the preceeding derivation rule, §2.3.3, p. 18.
[77] From the definition of an indirection node, §2.3.1, p. 13.
[78] This derivation rule, §4.5.3, p. 69.
[79] This and the preceeding derivation rule, from the metarule of §4.5.4, p. 78.

$$(\texttt{Equals}; C,\, a_1\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \nabla\ a_1'$$
$$\vdash\ (\texttt{Equals}; C,\, a_1'\!:\!S,\, G,\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \text{ not an indirection},\ a_2 \xrightarrow{G} \nabla\ a_2'$$
$$\vdash\ (\texttt{Equals}; C,\, a_1\!:\!a_2'\!:\!S,\, G,\, D)$$ [80]

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Embed}{-}\tau\ a_1',\ a_2 \xrightarrow{G} \mathsf{Embed}{-}\tau\ a_2'$$
$$\vdash\ (\texttt{Equals}; C,\, a_1'\!:\!a_2'\!:\!S,\, G,\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ a_1',\ a_2 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ a_2',\ \tau_1 \neq \tau_2$$
$$\vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Bool\ false}],\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Atom}\ \alpha,\ a_2 \xrightarrow{G} \mathsf{Atom}\ \alpha$$
$$\vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Bool\ true}],\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Atom}\ \alpha_1,\ a_2 \xrightarrow{G} \mathsf{Atom}\ \alpha_2,\ \alpha_1 \neq \alpha_2$$
$$\vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Bool\ false}],\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Bool}\ \beta,\ a_2 \xrightarrow{G} \mathsf{Bool}\ \beta$$
$$\vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Bool\ true}],\, D)$$

$$(\texttt{Equals}; C,\, a_1\!:\!a_2\!:\!S,\, G,\, D),\ a_1 \xrightarrow{G} \mathsf{Bool}\ \beta_1,\ a_2 \xrightarrow{G} \mathsf{Bool}\ \beta_2,\ \beta_1 \neq \beta_2$$
$$\vdash\ (C,\, a\!:\!S,\, G[a \mapsto \mathsf{Bool\ false}],\, D)$$

$$(\texttt{Equals}; C,\, a\!:\!b\!:\!S,\, G,\, D)\ \ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2,\ b \xrightarrow{G} \mathsf{Pair}\ b_1\ b_2$$
$$\vdash\ (\texttt{Unwind},\, s\!:\![\,],\, G \left[\begin{array}{rl} s & \mapsto \mathsf{Ap}\ s'\ u \\ s' & \mapsto \mathsf{Ap}\ s''\ t \\ s'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_And} \\ t & \mapsto \mathsf{Ap}\ t'\ a_1 \\ t' & \mapsto \mathsf{Ap}\ t''\ b_1 \\ t'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_Equals} \\ u & \mapsto \mathsf{Ap}\ u'\ a_2 \\ u' & \mapsto \mathsf{Ap}\ u''\ b_2 \\ u'' & \mapsto \mathsf{Fun}\ 2\ \mathsf{\_Equals} \end{array}\right],\ \langle S, C\rangle\!:\!D)$$ [81]

$$(\texttt{Equals}; C,\, a\!:\!b\!:\!S,\, G,\, D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a',\ b \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope}{-}\mathsf{Hu}\ b'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ b,\, a\!:\!b\!:\!S,\, G \left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & p_1 \mapsto \mathsf{RjnPtr}\ e \\ \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' & p_2 \mapsto \mathsf{RjnPtr}\ e \\ \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 & p_3 \mapsto \mathsf{RjnPtr}\ e \\ b_1 \mapsto \mathsf{Embed}{-}\mathsf{a}\ b_1' & b_1' \mapsto \mathsf{Logic\ Var}\ \ell_1 \\ b_2 \mapsto \mathsf{Embed}{-}\mathsf{b}\ b_2' & b_2' \mapsto \mathsf{Logic\ Var}\ \ell_2 \\ b_3 \mapsto \mathsf{Embed}{-}\mathsf{p}\ b_3' & b_3' \mapsto \mathsf{Pair}\ r_1\ r_2 \\ r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\ \ell_1 \mapsto \mathsf{Scope}{-}\mathsf{Atoms}\ p_1\ I & \\ \ell_2 \mapsto \mathsf{Scope}{-}\mathsf{Bools}\ p_2\ I & \\ \ell_3 \mapsto \mathsf{Scope}{-}\mathsf{Hu}\ \ell_4\ I & I \mapsto \mathsf{Ineqs}{-}\mathsf{End} \\ \ell_4 \mapsto \mathsf{Scope}{-}\mathsf{Hu}\ p_3\ I & id \mapsto \mathsf{UID} \\ e \mapsto \mathsf{Rejoin}\ 3\ id\ b'\ p_1\ p_2\ p_3 & \\ b \mapsto \mathsf{Split}\ 3\ id\ b_1\ b_2\ b_3 & \end{array}\right],\ D)$$

$(\mathtt{Equals}; C,\ a{:}b{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ b \xrightarrow{G} \mathsf{Embed{-}}\tau\ b',\ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$

$$\vdash (\mathtt{PropSplitUp}\ id\ a,\ a{:}b{:}S,\ G\left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & p_1 \mapsto \mathsf{RjnPtr}\ e \\ \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell'' & p_2 \mapsto \mathsf{RjnPtr}\ e \\ \ell'' \mapsto \mathsf{Union}\ \ell_2\ \ell_3 & p_3 \mapsto \mathsf{RjnPtr}\ e \\ a_1 \mapsto \mathsf{Embed{-}a}\ a_1' & a_1' \mapsto \mathsf{Logic\ Var}\ \ell_1 \\ a_2 \mapsto \mathsf{Embed{-}b}\ a_2' & a_2' \mapsto \mathsf{Logic\ Var}\ \ell_2 \\ a_3 \mapsto \mathsf{Embed{-}p}\ a_3' & a_3' \mapsto \mathsf{Pair}\ r_1\ r_2 \\ r_1 \mapsto \mathsf{Logic\ Var}\ \ell_3 & r_2 \mapsto \mathsf{Logic\ Var}\ \ell_4 \\ \ell_1 \mapsto \mathsf{Scope{-}Atoms}\ p_1\ I & \\ \ell_2 \mapsto \mathsf{Scope{-}Bools}\ p_2\ I & \\ \ell_3 \mapsto \mathsf{Scope{-}Hu}\ \ell_4\ I & id \mapsto \mathsf{UID} \\ \ell_4 \mapsto \mathsf{Scope{-}Hu}\ p_3\ I & I \mapsto \mathsf{Ineqs{-}End} \\ e \mapsto \mathsf{Rejoin}\ 3\ id\ a'\ p_1\ p_2\ p_3 & \\ a \mapsto \mathsf{Split}\ 3\ id\ a_1\ a_2\ a_3 & \end{array}\right], D)$$ [82]

$(\mathtt{Equals}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Bools}\ a'\ I$

$$\vdash (\mathtt{PropSplitUp}\ id\ a,\ a{:}S,\ G\left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell_2 \\ a_1 \mapsto \mathsf{Bool}\ \mathbf{true} & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Bool}\ \mathbf{false} & p_2 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \mathsf{UID} \end{array}\right], D)$$ [83]

$(\mathtt{Equals}{:} C,\ a{:}b{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ b \xrightarrow{G} \mathsf{Atom}\ \alpha,\ \ell \xrightarrow{G} \mathsf{Scope{-}Atoms}\ a'\ I$

$\vdash (\mathtt{Search\text{-}Ineq}\ \alpha\ a,\ I{:}[\,],\ G,\ \langle a{:}b{:}S, C\rangle{:}D)$

$(\mathtt{Equals}{:} C,\ a{:}b{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Atom}\ \alpha,\ b \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Atoms}\ b'\ I$

$\vdash (\mathtt{Search\text{-}Ineq}\ \alpha\ b,\ I{:}[\,],\ G,\ \langle a{:}b{:}S, C\rangle{:}D)$ [84]

$(\mathtt{Equals}; C,\ a_1{:}a_2{:}\ldots{:}a_n,\ G,\ D),\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2$

$\vdash (\mathtt{Search\text{-}Diseq}\ \ell_1\ \ell_2\ 1,\ a_n{:}[\,],\ G,\ \langle a_1{:}\ldots{:}a_n, C\rangle{:}D)$ [85]

$(\mathtt{Equals}; C,\ a_n^i{:}S,\ G,\ D),\ i \in \{1,\ \ldots,\ N_m\},$

$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$

$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$

$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$

$e_0 \xrightarrow{G} S\ u_1\ \ldots\ u_\phi,\ S \ne \mathsf{RjnPtr}$

$$\vdash (\mathtt{Equals}; C,\ a_n^i{:}S,\ G\left[\begin{array}{l} a_1^j \mapsto S\ r'_{1,j}\ \ldots\ r'_{\phi,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1} \end{array}\right], D)$$

$(\mathtt{Equals}; C,\ w{:}a_n^i{:}S,\ G,\ D),\ i \in \{1,\ \ldots,\ N_m\},$

w does not point to a RjnPtr-tagged node,

$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ 1 \le m \le n,\ 1 \le j \le N_m,$

$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ 2 \le m \le n,\ \text{some } k,\ 1 \le k \le N_{m-1},$

$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$

$e_0 \xrightarrow{G} S\ u_1\ \ldots\ u_\phi,\ S \ne \mathsf{RjnPtr}$

$$\vdash (\mathtt{Equals}; C,\ w{:}a_n^i{:}S,\ G\left[\begin{array}{l} a_1^j \mapsto S\ r'_{1,j}\ \ldots\ r'_{\phi,j},\ 1 \le j \le N_1 \\ r'_{i,j} \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r'_{i,1}\ \ldots\ r'_{i,N_1} \end{array}\right], D)$$ [86]

A.9.4 The And Command

The `And` command is defined by the following clauses:

$$(\texttt{And}; C,\ a_1{:}\,a_2{:}\,S,\ G,\ D),\ \ a_1 \xrightarrow{G} \mathsf{Bool\ true}$$
$$\vdash\ (\texttt{Unwind},\ a_2{:}\,[\,],\ G,\ \langle S, \texttt{And}; C\rangle{:}\,D)$$
$$(\texttt{And}; C,\ a_1{:}\,a_2{:}\,S,\ G,\ D),\ \ a_1 \xrightarrow{G} \mathsf{Bool\ false}$$
$$\vdash\ (\texttt{Unwind},\ a_1{:}\,[\,],\ G,\ \langle S, \texttt{And}; C\rangle{:}\,D)^{87}$$
$$(\texttt{And}; C,\ a_1{:}\,a_2{:}\,S,\ G,\ D),\ \ a_1 \xrightarrow{G} \nabla\ a_1'$$
$$\vdash\ (\texttt{And}; C,\ a_1'{:}\,a_2{:}\,S,\ G,\ D)^{88}$$
$$(\texttt{And}; C,\ a{:}\,S,\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ \ell \xrightarrow{G} \mathsf{Scope{-}Hu}\ a'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\ a{:}\,S,\ G \left[\begin{array}{ll} \ell' \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & \ell' \mapsto \mathsf{Union}\ \ell_1\ \ell_2 \\ a_1 \mapsto \mathsf{Bool}\ \mathbf{true} & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Bool}\ \mathbf{false} & p_2 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & id \mapsto \mathsf{UID} \end{array}\right],\ D)^{89}$$
$$(\texttt{And}; C,\ a_n^i{:}\,b{:}\,S,\ G,\ D),\ \ i \in \{1,\ \ldots,\ N_m\},$$
$$a_m^j \xrightarrow{G} \mathsf{RjnPtr}\ e_m,\ \ 1 \le m \le n,\ \ 1 \le j \le N_m,$$
$$e_m \xrightarrow{G} \mathsf{Rejoin}\ N_m\ id_m\ a_{m-1}^k\ a_m^1\ \ldots\ a_m^{N_m},\ \ 2 \le m \le n,\ \text{some}\ k,\ 1 \le k \le N_{m-1},$$
$$e_1 \xrightarrow{G} \mathsf{Rejoin}\ N_1\ id_1\ e_0\ a_1^1\ \ldots\ a_1^{N_1}$$
$$e_0 \xrightarrow{G} S\ u_1\ \ldots\ u_\phi,\ \ S \neq \mathsf{RjnPtr}$$
$$\vdash\ (\texttt{And}; C,\ a_n^i{:}\,b{:}\,S,\ G \left[\begin{array}{l} a_1^j \mapsto S\ r_{1,j}'\ \ldots\ r_{\phi,j}',\ 1 \le j \le N_1 \\ r_{i,j}' \mapsto \mathsf{RjnPtr}\ r_i,\ 1 \le i \le \phi,\ 1 \le j \le N_1 \\ r_i \mapsto \mathsf{Rejoin}\ N_1\ id_1\ u_i\ r_{i,1}'\ \ldots\ r_{i,N_1}' \end{array}\right],\ D)^{90}$$

A.10 Narrowing and Quasisharing Delimiters

The following commands perform narrowing manipulations and the propagation of quasisharing delimiters.

A.10.1 The PropSplitUp Command

The `PropSplitUp` command[91] is defined by the following clause:

[80] This and the preceeding derivation rule from the definition of an indirection node, §2.3.1, p. 13.
[81] This and the six preceeding derivation rules, §2.3.3, p. 19cmd:equalsEnd.
[82] This and the preceeding derivation rule, §4.5.3, p. 69.
[83] This derivation rule, §4.5.3, p. 70.
[84] This and the preceeding derivation rule, §4.5.3, p. 70.
[85] This and the preceeding derivation rule, §4.5.3, p. 71.
[86] This and the preceeding derivation rule, from the metarule of §4.5.4, p. 78.
[87] This derivation rule, §2.3.3, p. 19.
[88] From the definition of an indirection node, §2.3.1, p. 13.
[89] This derivation rule, §4.5.3, p. 69.
[90] This derivation rule, from the metarule of §4.5.4, p. 78.
[91] This command, §4.5.4, p. 72-77.

$$(\texttt{PropSplitUp}\ id\ s,\ a{:}[\,],G,\ \langle S,C\rangle{:}D),\ a \xrightarrow{G} \mathsf{Ex\text{-}Scope}\ a'\ r,$$
$$r \xrightarrow{G} \mathsf{Rejoin}\ 2\ id\ s\ r_1\ r_2,\ s \xrightarrow{G} \mathsf{Split}\ 2\ id\ s_1\ s_2,$$
$$r_1 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ r$$
$$\vdash\ (C,\ a'{:}S,\ G\left[\begin{array}{l} a \mapsto \nabla\ a' \\ r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \end{array}\right],\ D)$$

$$(\texttt{PropSplitUp}\ id\ s,\ a{:}[\,],G,\ \langle S,C\rangle{:}D),\ a \xrightarrow{G} \mathsf{Ex\text{-}Scope}\ a'\ r,$$
$$r \xrightarrow{G} \mathsf{Rejoin}\ 3\ id\ s\ r_1\ r_2\ r_3,\ s \xrightarrow{G} \mathsf{Split}\ 3\ id\ s_1\ s_2\ s_3,$$
$$r_1 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ r,\ r_3 \xrightarrow{G} \mathsf{RjnPtr}\ r$$
$$\vdash\ (C,\ a{:}S,\ G\left[\begin{array}{ll} r_1 \mapsto \nabla\ s_1 & r_2 \mapsto \nabla\ s_2 \\ r_3 \mapsto \nabla\ s_3 & a \mapsto \nabla\ a' \end{array}\right],\ D)$$

$$(\texttt{PropSplitUp}\ id\ a',\ a{:}[\,],G,\ \langle S,C\rangle{:}D),\ a \xrightarrow{G} \mathsf{Rejoin}\ 2\ id\ a'\ r_1\ r_2,$$
$$a' \xrightarrow{G} \mathsf{Split}\ 2\ id\ s_1\ s_2,\ r_1 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ a$$
$$\vdash\ (C,\ S,\ G\left[\begin{array}{l} r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \end{array}\right],\ D)$$

$$(\texttt{PropSplitUp}\ id\ a',\ a{:}[\,],G,\ \langle S,C\rangle{:}D),$$
$$a \xrightarrow{G} \mathsf{Rejoin}\ 3\ id\ a'\ r_1\ r_2\ r_3,\ a' \xrightarrow{G} \mathsf{Split}\ 3\ id\ s_1\ s_2\ s_3,$$
$$r_1 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_2 \xrightarrow{G} \mathsf{RjnPtr}\ a,\ r_3 \xrightarrow{G} \mathsf{RjnPtr}\ a$$
$$\vdash\ (C,\ S,\ G\left[\begin{array}{l} r_1 \mapsto \nabla\ s_1 \\ r_2 \mapsto \nabla\ s_2 \\ r_3 \mapsto \nabla\ s_3 \end{array}\right],\ D)$$

$$(\texttt{PropSplitUp}\ id\ e,\ x_0{:}\ldots{:}x_m{:}a_1{:}\ldots{:}a_n{:}e_n{:}[\,],G,\ D), e_0 \xrightarrow{G} \mathsf{Fun}\ n\ C$$
$$e_i \xrightarrow{G} \mathsf{Ap}\ e_{i-1}\ a_n,\ 1 \le i \le n$$
$$\vdash\ (\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],G,\ D)$$

$$(\texttt{PropSplitUp}\ id\ e_0,\ e_1{:}\ldots{:}e_n,\ G,\ D),\ e_i \xrightarrow{G} \mathsf{Ap}\ e_{i-1}\ a_i$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0,\ e_1{:}\ldots{:}e_n,\ G,\ D)$$

$$(\texttt{PropSplitUp}\ id\ a',\ x_0{:}\ldots{:}x_n{:}a{:}[\,],G,\ \langle S,C\rangle{:}D),$$
$$a \xrightarrow{G} M\ a',\ a' \xrightarrow{G} \mathsf{Split}\ n\ id\ a'_1\ \ldots\ a'_m$$
$$M \in \{\ \begin{array}{l} \mathsf{Embed}{-}\tau, \\ \mathsf{Singleton}, \\ \mathsf{Scope}{-}S\ \_\ I, \\ \mathsf{Diseq}\ s_1\ s_2, \\ \mathsf{Rejoin}\ \nu\ id\ \_\ s_1\ \ldots\ s_\nu \end{array}\ \}$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ a,\ S,\ G\left[\begin{array}{l} a \xrightarrow{G} \mathsf{Split}\ n\ id\ a_1\ \ldots\ a_m \\ a_i \xrightarrow{G} M\ a'_i, 1 \le i \le m \end{array}\right],\ D)$$

$$(\texttt{PropSplitUp}\ id\ a_1,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ \langle S,C\rangle{:}D),$$

$$e_n \xrightarrow{G} M\ a_1\ a_2,\ a_1 \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ a_1 \neq a_2,\ a_2 \text{ not an indirection}$$

$$M \in \{\ \mathsf{Pair},\ \mathsf{Union},\ \mathsf{Split}\ \nu\ id',\ id \neq id'\ \}$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\begin{bmatrix} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' & \\ e' \mapsto M\ a'\ r_1 & r_1 \mapsto \mathsf{RjnPtr}\ r \\ e'' \mapsto M\ a''\ r_2 & r_2 \mapsto \mathsf{RjnPtr}\ r \\ r \mapsto \mathsf{Rejoin}\ 2\ id\ a_2\ r_1\ r_2 & \end{bmatrix},\ D)$$

$$(\texttt{PropSplitUp}\ id\ a_2,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ \langle S,C\rangle{:}D),$$

$$e_n \xrightarrow{G} M\ a_1\ a_2,\ a_2 \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ a_1 \neq a_2,\ a_1 \text{ not an indirection}$$

$$M \in \{\ \mathsf{Pair},\ \mathsf{Union},\ \mathsf{Split}\ \nu\ id',\ id \neq id'\ \}$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\begin{bmatrix} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' & \\ e' \mapsto M\ r_1\ a' & r_1 \mapsto \mathsf{RjnPtr}\ r \\ e'' \mapsto M\ r_2\ a'' & r_2 \mapsto \mathsf{RjnPtr}\ r \\ r \mapsto \mathsf{Rejoin}\ 2\ id\ a_1\ r_1\ r_2 & \end{bmatrix},\ D)$$

$$(\texttt{PropSplitUp}\ id\ e,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ e_n \xrightarrow{G} M\ a_1\ a_2, a_1 \xrightarrow{G} \nabla\ a'$$

$$M \in \{\ \mathsf{Pair},\ \mathsf{Union},\ \mathsf{Split}\ \nu\ id',\ id \neq id'\ \}$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ e,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_n \mapsto M\ a'\ a_2],\ D)$$

$$(\texttt{PropSplitUp}\ id\ e,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ e_n \xrightarrow{G} M\ a_1\ a_2,$$

$$a_1 \text{ not an indirection},\ a_2 \xrightarrow{G} \nabla\ a'$$

$$M \in \{\ \mathsf{Pair},\ \mathsf{Union},\ \mathsf{Split}\ \nu\ id',\ id \neq id'\ \}$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ e,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_n \mapsto M\ a_1\ a'],\ D)$$

$$(\texttt{PropSplitUp}\ id\ a,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ \langle S,C\rangle{:}D),$$

$$a \xrightarrow{G} \mathsf{Split}\ 2\ id\ a'\ a'',\ e_n \xrightarrow{G} M\ a\ a$$

$$M \in \{\ \mathsf{Pair},\ \mathsf{Union},\ \mathsf{Split}\ \nu\ id',\ id \neq id'\ \}$$

$$\vdash\ (\texttt{PropSplitUp}\ id\ e_n,\ S,\ G\begin{bmatrix} e_n \mapsto \mathsf{Split}\ 2\ id\ e'\ e'' \\ e' \mapsto M\ a'\ a' \\ e'' \mapsto M\ a''\ a'' \end{bmatrix},\ D)$$

A.10.2 The `InitSpineSplit` Command

The `InitSpineSplit` command[92] is defined by the following clause:

$$(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,\ a \xrightarrow{G} \nabla\ a'$$

$$\vdash\ (\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G[e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a'],\ D)$$

[92]This command, §4.5.4, p. 74-75.

$$(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
$$a \xrightarrow{G} \mathsf{Split}\ m\ id\ s_1\ \ldots\ s_m$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_1, e_2{:}\ldots{:}e_n{:}[\,], G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ m\ id\ e'_1\ \ldots\ e'_m \\ e'_i \mapsto \mathsf{Ap}\ f\ s_i,\ 1 \le i \le m \end{array}\right], D)$$

$$(\texttt{InitSpineSplit}\ id,\ e_1{:}\ldots{:}e_n{:}[\,],\ G,\ D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$

a is a pointer neither to a Split node identified by id, nor to an indirection node

$$\vdash\ (\texttt{SpineSplit}\ id\ e_1, e_2{:}\ldots{:}e_n{:}[\,], G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_m \\ e'_i \mapsto \mathsf{Ap}\ f\ r_i,\ 1 \le i \le m \\ r_i \mapsto \mathsf{RjnPtr}\ r,\ 1 \le i \le m \\ r \mapsto \mathsf{Rejoin}\ n\ id\ a\ r_1\ \ldots\ r_m \end{array}\right], D)$$

A.10.3 The `SpineSplit` Command

The `SpineSplit` command[93] is defined by the following clause:

$$(\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_n{:}[\,], G, D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,\ \ f \xrightarrow{G} \nabla\ f'$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_n{:}[\,], G[e_1 \xrightarrow{G} \mathsf{Ap}\ f'\ a'], D)$$

$$(\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_n{:}[\,], G, D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ e_0\ a,\ \ a \xrightarrow{G} \nabla\ a',$$
$$e_0 \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_m$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_n{:}[\,], G[e_1 \xrightarrow{G} \mathsf{Ap}\ e_0\ a'], D)$$

$$(\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_m{:}[\,], G, D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
$$f \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_n,\ \ a \xrightarrow{G} \mathsf{Split}\ n\ id\ s_1\ \ldots\ s_n$$
$$\vdash\ (\texttt{SpineSplit}\ id\ e_1, e_2{:}\ldots{:}e_m{:}[\,], G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_n \\ e'_i \mapsto \mathsf{Ap}\ f_i\ s_i,\ 1 \le i \le n \end{array}\right], D)$$

$$(\texttt{SpineSplit}\ id\ e_0, e_1{:}\ldots{:}e_m{:}[\,], G, D),\ \ e_1 \xrightarrow{G} \mathsf{Ap}\ f\ a,$$
$$f \xrightarrow{G} \mathsf{Split}\ n\ id\ f_1\ \ldots\ f_n,$$

a is a pointer neither to a Split node identified by id, nor to an indirection node unless that indirection is to a Rejoin node

$$\vdash\ (\texttt{SpineSplit}\ id\ e_1, e_2{:}\ldots{:}e_m{:}[\,], G\left[\begin{array}{l} e_1 \mapsto \mathsf{Split}\ n\ id\ e'_1\ \ldots\ e'_n \\ e'_i \mapsto \mathsf{Ap}\ f_i\ r_i,\ 1 \le i \le n \\ r_i \mapsto \mathsf{RjnPtr}\ r,\ 1 \le i \le n \\ r \mapsto \mathsf{Rejoin}\ n\ id\ a\ r_1\ \ldots\ r_n \end{array}\right], D)$$

$$(\texttt{SpineSplit}\ id\ e_n, [\,], G, \langle S, C\rangle{:}D)\ \ \vdash\ \ (\texttt{PropSplitUp}\ id\ e_n, S, G, D)$$

A.10.4 The `Remove` Command

The `Remove` command[94] is defined by the following clause:

$$(\texttt{Remove}; C_0, a{:}[\,], G, \langle S, c; C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ a'\ a_1\ \ldots\ a_N,$$
$$a_i \xrightarrow{G} \mathsf{RjnPtr}\ a$$
$$\vdash\ (C, S, G\left[\begin{array}{l} a_i \mapsto \nabla\ a' \\ a \mapsto \nabla\ a' \end{array}\right], D)$$

[93] This command, §4.5.4, p. 75-75.
[94] This command, §4.5.4, p. 77.

A.10.5 The `PerformUpdate` Command

The `PerformUpdate` command[95] is defined by the following clause:

$$(\texttt{PerformUpdate}, a{:}[\,], G, D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} \nabla\ e$$
$$\vdash\ (\texttt{PerformUpdate}, a{:}[\,], G\,[a \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ e]\,, D)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} \mathsf{Logic\ Var}\ \alpha$$
$$\vdash\ (C,\ S,\ G\,[a \mapsto \nabla\ \beta]\,,\ D)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} N,$$
$$N \in\ \{\mathsf{Phi}, \mathsf{Atom}\ i, \mathsf{Bool}\ b, \mathsf{Logic\ Var}\ \alpha',\ \alpha \neq \alpha'\}$$
$$\vdash\ (C,\ S,\ G\,[a \mapsto \nabla\ a']\,,\ D)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',$$
$$a' \xrightarrow{G} \mathsf{RjnPtr}\ r\ r \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ r'\ r_1\ \ldots\ r_N$$
$$\vdash\ \left(C,\ S,\ G\begin{bmatrix} a \mapsto \nabla\ a' \\ r \mapsto \mathsf{Rejoin}\ n\ id\ s\ r_1\ \ldots\ r_N \\ s \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ r' \end{bmatrix},\ D\right)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} N\ a_1$$
N the tag of a node with one child in the graph, but not an indirection node
$$\vdash\ \left(C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \end{bmatrix},\ D\right)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} N\ a_1\ a_2$$
N the tag of a node with two children in the graph
$$\vdash\ \left(C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1'\ a_2' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \\ a_2' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_2 \end{bmatrix},\ D\right)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G,\ \langle S,C\rangle{:}\,D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} N\ a_1\ a_2\ a_3$$
N the tag of a node with three children in the graph
$$\vdash\ \left(C,\ S,\ G\begin{bmatrix} a \mapsto N\ a_1'\ a_2'\ a_3' \\ a_1' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_1 \\ a_2' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_2 \\ a_3' \mapsto \mathsf{LVUpdate}\ \alpha\ \beta\ a_3 \end{bmatrix},\ D\right)$$

$$(\texttt{PerformUpdate}, a{:}[\,], G, D),\ a \xrightarrow{G} \mathsf{LVUpdate}\ \alpha\ \beta\ a',\ a' \xrightarrow{G} \mathsf{LVUpdate}\ \alpha'\ \beta'\ a''$$
$$\vdash\ (\texttt{PerformUpdate}, a'{:}[\,], G,\ \langle a{:}[\,], \texttt{PerformUpdate}\rangle{:}\,D)$$

A.11 Inequality Relationships

This section described commands used at run-time to manage inequality information.

A.11.1 The `Search-Ineq` Command

The `Search-Ineq` command[96] is defined by the following clauses:

[95] This command, §4.5.4, p. 80.
[96] This command, §4.5.3, p. 70.

$$(\texttt{Search-Ineq}\ \alpha\ u,\ I{:}[\,],\ G,\ \langle a_1{:}a_2{:}S,C\rangle{:}D),\ I \xrightarrow{G} \mathsf{Ineq}\ \alpha\ I'$$
$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Bool\ false}],\ D)$$
$$(\texttt{Search-Ineq}\ \alpha\ u,\ I{:}[\,],\ G,\ D),\ I \xrightarrow{G} \mathsf{Ineq}\ \alpha'\ I',\ \alpha \neq \alpha'$$
$$\vdash\ (\texttt{Search-Ineq}\ \alpha\ u,\ I'{:}[\,],\ G,\ D)$$
$$(\texttt{Search-Ineq}\ \alpha\ u,\ I_0{:}[\,],\ G,\ \langle S,C\rangle{:}D),$$
$$I_0 \xrightarrow{G} \mathsf{Ineqs{-}End},\ u \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ell \xrightarrow{G} \mathsf{Scope{-}Atoms}\ u'\ I$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ u,\ S,\ G\left[\begin{array}{ll} \ell \mapsto \mathsf{Ex\text{-}Scope}\ \ell'\ e & I' \mapsto \mathsf{Ineq}\ \alpha\ I \\ \ell' \mapsto \mathsf{Union}\ p_1\ \ell_2 & u_1 \mapsto \mathsf{Atom}\ \alpha \\ \ell_2 \mapsto \mathsf{Scope{-}Atoms}\ p_2\ I' & u_2 \mapsto \mathsf{Logic\ Var}\ \ell_2 \\ p_1 \mapsto \mathsf{RjnPtr}\ e & p_2 \mapsto \mathsf{RjnPtr}\ e \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ u'\ p_1\ p_2 & \\ u \mapsto \mathsf{Split}\ 2\ id\ u_1\ u_2 & id \mapsto \mathsf{UID} \end{array}\right],\ D)$$

A.11.2 The `Search-Diseq` Command

The `Search-Diseq` command[97] is defined by the following clauses:

$$(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ \ell_1{:}S_0,\ G,\ D),\ \ell_1 \xrightarrow{G} \mathsf{Scope{-}}S\ a'\ I,\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1$$
$$D = \langle a{:}b{:}S,C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle s_1{:}\ldots{:}s_m{:}\ell_1{:}[\,],C_0\rangle{:}D''$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ \ell_1,\ a{:}b{:}S,\ G\left[\begin{array}{ll} \ell_1 \mapsto \mathsf{Ex\text{-}Scope}\ \ell_1'\ \ell' & \ell_1' \mapsto \mathsf{Union}\ p_1\ \ell_b \\ a_1 \mapsto \mathsf{Logic\ Var}\ \ell_2 & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Logic\ Var}\ \ell_b & \ell_b \mapsto \mathsf{Scope{-}}S\ \ell_b'\ I \\ id \mapsto \mathsf{UID} & \ell_b' \mapsto \mathsf{Diseq}\ \ell_b\ \ell_2\ p_2 \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & p_2 \mapsto \mathsf{RjnPtr}\ e \end{array}\right],\ D')$$
$$(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ \ell_2{:}S_0,\ G,\ D),\ \ell_2 \xrightarrow{G} \mathsf{Scope{-}}S\ a\ I$$
$$D = \langle a{:}b{:}S,C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle s_1{:}\ldots{:}s_m{:}\ell_2{:}[\,],C_0\rangle{:}D''$$
$$\vdash\ (\texttt{PropSplitUp}\ id\ \ell_2,\ a{:}b{:}S,\ G\left[\begin{array}{ll} \ell_2 \mapsto \mathsf{Ex\text{-}Scope}\ \ell_2'\ \ell' & \ell_2' \mapsto \mathsf{Union}\ p_1\ \ell_b \\ a_1 \mapsto \mathsf{Logic\ Var}\ \ell_1 & p_1 \mapsto \mathsf{RjnPtr}\ e \\ a_2 \mapsto \mathsf{Logic\ Var}\ \ell_b & \ell_b \mapsto \mathsf{Scope{-}}S\ \ell_b'\ I \\ id \mapsto \mathsf{UID} & \ell_b' \mapsto \mathsf{Diseq}\ \ell_b\ \ell_1\ p_2 \\ e \mapsto \mathsf{Rejoin}\ 2\ id\ a'\ p_1\ p_2 & \\ a \mapsto \mathsf{Split}\ 2\ id\ a_1\ a_2 & p_2 \mapsto \mathsf{RjnPtr}\ e \end{array}\right],\ D')$$
$$(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ S_0,\ G,\ D),$$
$$D = \langle a_1{:}a_2{:}S,C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle a{:}[\,],C_0\rangle{:}D''$$
$$a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a',\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2$$
$$\vdash\ (C,\ b{:}S,\ G[b \mapsto \mathsf{Bool\ false}],\ D)$$
$$(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ S_0,\ G,\ D),$$
$$D = \langle a_2{:}a_1{:}S,C\rangle{:}D',\ D' = d_1{:}\ldots{:}d_{n-1}{:}\langle a{:}[\,],C_0\rangle{:}D''$$
$$a \xrightarrow{G} \mathsf{Diseq}\ a_1\ a_2\ a',\ a_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_1,\ a_2 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell_2$$
$$\vdash\ (C,\ b{:}S,\ G[b \mapsto \mathsf{Bool\ false}],\ D)$$

[97]This command, §4.5.3, p. 71.

$$(\texttt{Search-Diseq}\ \ell_1\ \ell_2\ n,\ S,\ G,\ D),\ \ D = d_0{:}\ldots{:}d_{n-1}{:}\langle a_0{:}\ldots a_j{:}[\,], C_0\rangle{:}D',$$

$$a_j \xrightarrow{G} N,\ \ N \notin \ \{\mathsf{Logic\ Var}\ \ell_1,\ \mathsf{Logic\ Var}\ \ell_2,\ \mathsf{Diseq}\ a_1\ a_2\ a',\ \mathsf{Diseq}\ a_2\ a_1\ a'\ \}$$

$$\vdash\ (\texttt{Search-Diseq}\ \ell_1\ \ell_2\ (n+1),\ S,\ G,\ D)$$

A.11.3 The `DiseqPreExamine` Command

The `DiseqPreExamine` command[98] is defined by the following clauses:

$$(\texttt{DiseqPreExamine};C,\ a{:}[\,],\ G,\ D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$e_1 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e_1',\ \ e_2 \xrightarrow{G} \mathsf{Embed}{-}\tau\ e_2'$$

$$\vdash\ (\texttt{DiseqPreExamine};C,\ a{:}[\,],\ G[a \mapsto \mathsf{Diseq}\ e_1'\ e_2'\ a'],\ D)$$

$$(\texttt{DiseqPreExamine};C_0,\ a{:}[\,],\ G,\ \langle S, C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$e_1 \xrightarrow{G} \mathsf{Embed}{-}\tau_1\ e_1',\ \ e_2 \xrightarrow{G} \mathsf{Embed}{-}\tau_2\ e_2',\ \ \tau_1 \neq \tau_2$$

$$\vdash\ (C,\ a'{:}S,\ G[a \mapsto \nabla\ a'],\ D)$$

$$(\texttt{DiseqPreExamine};C_0,\ a{:}[\,],\ G,\ \langle S, C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$e_1 \xrightarrow{G} \mathsf{Pair}\ p_1\ p_3,\ \ e_2 \xrightarrow{G} \mathsf{Pair}\ p_2\ p_4$$

$$\vdash\ (\texttt{Unwind};\texttt{Retry},\ a{:}[\,],\ G\left[\begin{array}{ll} t_1' \mapsto \mathsf{Ap}\ \gamma_e\ e_1 & t_1'' \mapsto \mathsf{Ap}\ \gamma_e\ e_2 \\ t_2' \mapsto \mathsf{Ap}\ t_1'\ e_3 & t_2'' \mapsto \mathsf{Ap}\ t_1''\ e_4 \\ n_2 \mapsto \mathsf{Ap}\ n_1\ t_2' & n_2 \mapsto \mathsf{Ap}\ \gamma_a\ t_2'' \\ a \mapsto \mathsf{Ap}\ a_2\ a' & a_2 \mapsto \mathsf{Ap}\ a_1\ \gamma_\phi \\ a_1 \mapsto \mathsf{Ap}\ \gamma_i\ n_2 & \gamma_\phi \mapsto \mathsf{Phi} \\ \gamma_a \mapsto \mathsf{Fun}\ 2\ \texttt{\_and} & \gamma_i \mapsto \mathsf{Fun}\ 3\ \texttt{\_if} \\ \gamma_e \mapsto \mathsf{Fun}\ 2\ \texttt{\_equals} & \end{array}\right],\ \langle S, C\rangle{:}D)$$

$$(\texttt{DiseqPreExamine};C_0,\ a{:}[\,],\ G,\ \langle S, c;C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$e_1 \xrightarrow{G} \mathsf{Logic\ Var}\ \ell,\ \ e_2 \xrightarrow{G} \mathsf{Atom}\ \alpha,\ \ \ell \xrightarrow{G} \mathsf{Scope{-}Atoms}\ e\ I$$

$$\vdash\ (C,\ a'{:}S,\ G\left[\begin{array}{l} a \mapsto \nabla\ a' \\ \ell \mapsto \mathsf{Scope{-}Atoms}\ e\ I' \\ I' \mapsto \mathsf{Ineq}\ I\ \alpha \end{array}\right],\ D)$$

$$(\texttt{DiseqPreExamine};C_0,\ a{:}[\,],\ G,\ \langle S, c;C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$e_1 \xrightarrow{G} N,\ \ e_2 \xrightarrow{G} N,\ \ N \in \{\mathsf{Atom}\ \alpha,\ \mathsf{Bool}\ \beta,\ \mathsf{Logic\ Var}\ \ell\}$$

$$\vdash\ (C,\ a{:}S,\ G[a \mapsto \mathsf{Phi}],\ D)$$

$$(\texttt{DiseqPreExamine};C_0,\ a{:}[\,],\ G,\ \langle S, C\rangle{:}D),\ \ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a',$$

$$\text{and}\ \left\{\begin{array}{c} \textit{-either-} \\ e_1 \xrightarrow{G} \mathsf{Atom}\ \alpha_1,\ \ e_2 \xrightarrow{G} \mathsf{Atom}\ \alpha_2,\ \ \alpha_1 \neq \alpha_2 \\ \textit{-or-} \\ e_1 \xrightarrow{G} \mathsf{Bool}\ \beta_1,\ \ e_2 \xrightarrow{G} \mathsf{Bool}\ \beta_2,\ \ \beta_1 \neq \beta_2 \end{array}\right\}$$

$$\vdash\ (C,\ a'{:}S,\ G[a \mapsto \nabla\ a'],\ D)$$

otherwise, $(\texttt{DiseqPreExamine};C,\ S,\ G,\ D) \vdash (C,\ S,\ G,\ D)$

A.12 Global Constants

Two commands are used to allocate and set up global constants.

[98] This command, §4.5.4, p. 79.

A.12.1 The `Alloc Globals` Command

The **`Alloc Globals`** command[99] is defined by the following clause:

$$(\texttt{Alloc Globals}\ n; C,\ S,\ G,\ D) \vdash (C,\ S,\ G[\gamma_0 \mapsto \mathsf{Hole},\ \ldots,\ \gamma_{n-1} \mapsto \mathsf{Hole}],\ D)$$

A.12.2 The `Set Global` Command

The **`Set Global`** command[100] is defined by the following clause:

$$(\texttt{Set Global}\ n\ k\ a; C,\ S,\ G,\ D) \vdash (C,\ S,\ G[\gamma_n \mapsto \mathsf{Fun}\ k\ a],\ D)$$

A.13 The `Print` Family of Commands

The **`Print`** family consists of commands for printing expressions, tags and literal strings.

A.13.1 The `Print` Command

The **`Print`** command is defined by the following clauses:

$$
\begin{array}{rl}
(O, \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Atom}\ \alpha & \vdash\ (O; \alpha,\ C,\ S,\ G,\ D) \\
(O, \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Bool}\ \beta & \vdash\ (O; \beta,\ C,\ S,\ G,\ D)^{101} \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Embed}{-}\tau\ a' & \vdash\ (\texttt{\_Print\_Tagged\_}\tau,\ a'{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}\,D) \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Pair}\ a_1\ a_2 & \vdash\ (\texttt{\_Print\_Pair},\ a_1{:}a_2{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}\,D)^{102} \\
(O, \texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Phi} & \vdash\ (O; \phi,\ C,\ S,\ G,\ D) \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Singleton}\ a' & \vdash\ (\texttt{\_print\_singleton},\ a'{:}[\,],\ G,\ \langle S, C\rangle{:}\,D)^{103} \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D) & \vdash\ (\texttt{\_print\_union},\ a_1{:}a_2{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}\,D)^{104} \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Scope}{-}S\ a'\ I & \\
 & \vdash\ (\texttt{\_print\_lv\_abstr},\ a{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}\,D) \\
(O, \texttt{Print}; C,\ a{:}S,\ G,\ D)\ a \xrightarrow{G} \mathsf{Logic\ Var}\ \ell & \vdash\ (O; \gamma(\ell),\ C,\ S,\ G,\ D) \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{Diseq}\ e_1\ e_2\ a' & \\
 & \vdash\ (\texttt{\_print\_diseq},\ e_1{:}e_2{:}a'{:}[\,],\ G,\ \langle S, \texttt{Print}; C\rangle{:}\,D) \\
(\texttt{Print}; C,\ a{:}S,\ G,\ D),\ a \xrightarrow{G} \mathsf{RjnPtr}\ a',\ a' \xrightarrow{G} \mathsf{Rejoin}\ n\ id\ e\ \ldots & \\
 & \vdash\ (\texttt{Print}; C,\ e{:}S,\ G,\ D)^{105}
\end{array}
$$

where the function γ is as in §A.13.4, and with auxiliary code sequence definitions

```
_Print_Embedded_τ:  Eval
                    Print
                    PrintTag τ
                    Return
```

and

[99] This command, §2.3.2, p. 17.
[100] This command, §2.3.2, p. 17.
[101] This and the preceeding derivation rule, §2.3.4, p. 19.
[102] This and the preceeding derivation rule, §2.3.4, p. 20.
[103] This and the preceeding derivation rule, §4.7, p. 83.
[104] This derivation rule, §4.7, p. 83.
[105] This and the three preceeding derivation rules, §4.7, p. 84.

```
        _Print_Pair:  Eval
                      Print
                      Eval
                      Print
                      Return
```

and

```
  _print_singleton:  PrintMesg {
                     Eval-wh
                     Eval-prw
                     Print
                     PrintMesg }
                     Return
```

and

```
   _print_union  Print
                 Query
                 Jfalse _exit
                 Eval-wh
                 Eval-prw
                 Print
                 Return
          _exit  Pop 1
                 Return
```

and

```
  _print_lv_abstr:  Push 0
                    GetScopedValue
                    Push 1
                    Push 0
                    PrintMesg U_[
                    PrintAddr
                    PrintMesg ∈
                    PrintRange
                    PrintMesg ]
                    Print
                    Pop 1
                    Return
```

and

```
     _print_diseq:  PrintMesg _[
                    Print
                    PrintMesg ≠
                    Print
                    PrintMesg ]
                    Print
```

A.13.2 The PrintTag Command

The **PrintTag** command[106] is defined by the following clause:

$$(O, \texttt{PrintTag}\ \ \tau; C, S, G, D) \vdash (O; \tau, C, S, G, D)$$

A.13.3 The PrintMesg Command

The **PrintMesg** command[107] is defined by the following clause:

$$(O, \texttt{PrintMesg}\ \ c; C, S, G, D)\ \vdash\ (O; c, C, S, G, D)$$

[106] This command, §2.3.4, p. 20.
[107] This command, §4.7, p. 83.

A.13.4 The `PrintAddr` Command

The **`PrintAddr`** command[108] is defined by the following clause:

$$(O, \texttt{PrintAddr}; C, a\!:\!S, G, D) \;\vdash\; (O; \gamma(a), C, S, G, D)$$

Where the function γ is taken to map addresses into some printable form.

A.13.5 The `PrintRange` Command

The **`PrintRange`** command[109] is defined by the following clauses:

$$(O, \texttt{PrintRange}; C, a\!:\!S, G, D),\; a \xrightarrow{G} \mathsf{Scope\!-\!Hu}\; a'\; I \;\vdash\; (O; \texttt{hu}, C, S, G, D)$$

$$(O, \texttt{PrintRange}; C, a\!:\!S, G, D),\; a \xrightarrow{G} \mathsf{Scope\!-\!Atoms}\; a'\; I_0,\; I_0 \xrightarrow{G} \mathsf{Ineq}\; I_1\; a_1,\; \ldots$$
$$I_i \xrightarrow{G} \mathsf{Ineq}\; I_{i+1}\; a_{i+1},\; \ldots I_n \xrightarrow{G} \mathsf{Ineqs\!-\!End}$$
$$\vdash\; (O; \texttt{atoms} \setminus \{A_{a_1},\; \ldots,\; A_{a_n}\}, C, S, G, D)$$

$$(O, \texttt{PrintRange}; C, a\!:\!S, G, D),\; a \xrightarrow{G} \mathsf{Scope\!-\!Bools}\; a'\; I \;\vdash\; (O; \texttt{bools}, C, S, G, D)$$

A.14 The `Query` Command

As described in §4.7, the **`Query`** command will be used without a formal deterministic definition. The command is described on p. 83. A non-deterministic (choice) definition can be made as follows:

$$(\texttt{Query}; C, S, G, D) \vdash (C, a\!:\!S, G\,[a \mapsto \mathsf{Bool\ true}]\,, D)$$
$$(\texttt{Query}; C, S, G, D) \vdash (C, a\!:\!S, G\,[a \mapsto \mathsf{Bool\ false}]\,, D)$$

[108] This command, §4.7, p. 83.
[109] This command, §4.7, p. 84.

Bibliography

[1] Werner Hans, Rita Loogen, and Stephan Winkler. On the interaction of lazy evaluation and backtracking.

[2] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2), 1989.

[3] 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.

[4] M. G. Main. A powerdomain primer. *Bulletin of the European Association for Theoretical Computer Science*, 33, October 1987.

[5] Juan José Moreno-Navarro and Mario Rodríguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–224, 1992.

[6] 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, Springer, 1992.

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

[8] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages*. International Series in Computer Science. Prentice Hall, 1992.

[9] Uday S. Reddy. Functional logic languages, part 1. In *Lecture Notes in Computer Science 279*. Workshop on Graph Reduction, Springer, 1987.

[10] 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.

[11] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, University of Oxford, 1971.

Biography

John Maraist was born on January 12, 1970, in Lafayette, Louisiana. Mr. Maraist received a B.S. degree in Computer Science and Mathematics from Tulane University in 1992, and a M.E. from Tulane in Computer Science in 1993. Mr. Maraist participated in Tulane's Junior Year Abroad program at the University of Glasgow, Scotland, is a member of the Mathematics honor society ΦΜΕ and is a charter member of the Tulane chapter of ΥΠΕ, the computer science honor society. Outside of the lab, John is found usually at WTUL, Tulane's radio station, where he was the General Manager for two years, or otherwise roaming around curious parts of New Orleans and the Pontchartrain North Shore.