Graph-based Operational Semantics of a Lazy Functional Language
by Kristoffer H. Rose
DIKU Semantics Note D–146

22

Graph-based Operational Semantics of a Lazy Functional Language

Kristoffer H. Rose

22.1 INTRODUCTION

This chapter proposes the use of term graph rewriting (TGR) as the basis for the specification formalism *graph operational semantics* (GOS) intended for lazy functional programming languages. We demonstrate how GOS is a compromise between theory and practice, i.e., formal solidity desirable in the design process, the detail required for implementation, and the compactness convenient for users, by using it to specify BAWL featuring the operational aspects of the language used in the standard text book *Introduction to Functional Programming* by Bird and Wadler [BW88].

But first we summarize BAWL and explain how it is susceptible to TGR; then we briefly sketch the background of this work before presenting an overview of the remainder of the chapter.

BAWL—a generic lazy functional programming language

Although it is not particularly large, *BAWL is not a toy language.* Consequently, we will not aim for a complete specification of all build-in operators, types, etc. of BAWL here, but just give an overview of the features particularly relevant to graph reduction.

Figure 22.1 shows a BAWL version of the "quicksort" algorithm as found in [BW88], and a list of integers to sort—thus evaluating the program should print the result list "$[1,2,3,4,5,6,7,8]$". To be precise, the program defines and uses the function *sort* with a local definition of the infix list concatenation operator $+\!\!+$, assumes standard infix $<$

Term Graph Rewriting: Theory and Practice, eds. M.R.Sleep, M.J.Plasmeijer and M.C. van Eekelen

$$
\begin{array}{l}
sort\ [\,] \qquad = [\,] \\
sort\ (x : xs) = sort\ [\ u \mid u \leftarrow xs; u < x\ ] +\!\!+ [x] +\!\!+ sort\ [\ u \mid u \leftarrow xs; u \geq x\ ] \\
\qquad\qquad \textbf{where} \quad [\,] +\!\!+ ys \qquad = ys \\
\qquad\qquad\qquad\quad\ (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \\
?\ sort\ [2,4,6,8,7,5,3,1]
\end{array}
$$

Figure 22.1 Quicksort in BAWL

and $\geq$ relations over the standard numeric type, constructors [ ] and infix : for lists of numbers, as well as standard list notation, notably "list comprehensions", i.e., the [_ | _] notation described in section 3.2 of [BW88] (originally called "ZF-expressions" in [Tur82]).

But we will not be concerned with such details here. Instead we will follow tradition and specify BAWL by giving a "Core BAWL" subset that is sufficiently powerful that all BAWL programs may be translated into it *linearly*, i.e., without duplication of code, and that has all the features that we wish to specify semantically. So Core BAWL is not a "low-level" language in the usual sense—in particular it includes full pattern matching. The only true simplification is that Core BAWL is *untyped* (to avoid the need for a static semantics): it just allows an unspecified set of constructors (each with a certain arity) and two special pattern forms for matching constructors. The first, $!x$, will only match arguments in weak head normal form (WHNF), i.e., arguments where the root is on a form that is not evaluable (it may be a constructor or an application with insufficient arguments). This can be used to code the function *strict* of [BW88] as $strict\ f\ !x = f\ x$. The second, (!! _ ... _), will match any constructor value with components matching the individual _ ... _ and can be used to code the *seq* function (using ; as a line delimiter): $seq\ (!!)\ x = x$; $seq\ (!!\ x_1)\ x = seq\ x_1\ x$; $seq\ (!!\ x_1\ x_2)\ x = seq\ x_1\ (seq\ x_2\ x)$, ... up to any finite constructor arity.

DEFINITION **22.1.1** *Let I range over the identifiers $\mathcal{I}$ and C over the constructors $\mathcal{C}$. The* Core BAWL programs *are defined inductively as follows ($X \ldots X$ means zero or more Xs):*

$$
\begin{array}{lrcl}
\textit{Core BAWL program}: & B & ::= & \begin{array}{l} S \\ ?\ E \end{array} \\
\textit{Script}: & S & ::= & \begin{array}{l} D \\ \vdots \\ D \end{array} \\
\textit{Definition}: & D & ::= & \begin{array}{l} I\ P \ldots P\ =\ E\ \textbf{where}\ S \\ \vdots \\ I\ P \ldots P\ =\ E\ \textbf{where}\ S \end{array} \\
\textit{Pattern}: & P & ::= & I \ \mid\ !\,I \ \mid\ (\,!!\ P \ldots P\,) \ \mid\ (\,C\ P \ldots P\,) \\
\textit{Expression}: & E & ::= & I \ \mid\ (\,C\ E \ldots E\,) \ \mid\ (\,E\ E \ldots E\,)
\end{array}
$$

where (1) the $I\ P \ldots P$ of each D should have the same I and number of Ps and (2) each C should be followed by the same number of Ps and Es everywhere.

In the rest of this chapter we will refer to this subset simply as BAWL; readers doubting that *sort* above can in fact be expressed in it may find the translation in figure 22.2 (a thorough explanation of the kind of translations we have used is given

in chapter 3, 5, and 7 of Peyton Jones's book [PJ87]). The only variation is that we compile guards and tests into pattern matching rather than vice versa (the details may be found in [Ros92b, Ros91]).

```
sort []       = []  where
sort ((:) x xs) = ((++) (sort (cont xs)) ((++) ((:) x []) (sort (cont' xs))))
  where  (++) [] ys         = ys  where
         (++) ((:) x xs) ys = ((:) x ((++) xs ys))  where
         cont []           = []  where
         cont ((:) u rest) = res
           where  res = (tmp ((<) u x))
                    where  tmp True  = ((:) u e')  where
                           tmp False = e'  where
                  e' = (cont rest)  where
         cont ((:) p rest) = (cont rest)  where
         cont' []          = []  where
         cont' ((:) u rest) = res
           where  res = (tmp ((>=) u x))
                    where  tmp True  = ((:) u e')
                           tmp False = e'
                  e' = (cont' rest)  where
         cont' ((:) p rest) = (cont' rest)  where
? (sort ((:) 2 ((:) 4 ((:) 6 ((:) 8 ((:) 7 ((:) 5 ((:) 3 ((:) 1 [])))))))))
```

Figure 22.2 Core BAWL version of *sort*

Evaluation of BAWL expressions by TGR

The top drawing of figure 22.3 depicts the term graph representation of the BAWL expression

$$[1] +\!\!+ [2] \quad \textbf{where} \quad \begin{array}{lcl} [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

that we might need to evaluate during the execution of the *sort* program above (ignore the dotted lines for now). Each of the numbered nodes in the graph represents a subterm of the kind indicated by its label: The @-nodes denote application of a function to some arguments; Λ_2-nodes have the alternative (uniform) equations as subgraphs; and λ-nodes have the LHS and RHS (i.e., the left and right hand sides) of a single equation as left and right subgraphs[1] where ■ denotes the position of the function itself.

The main difference between this representation and a traditional abstract syntax tree is that *variables are represented by pointers to their value*; in particular unbound variables are represented by pointers to the ?-node in the representation of the LHS where the variable is defined. For example, in the first equation the variable *ys* is represented by node *15*, in the second x is represented by node *20*, *xs* by *21*, and *ys* by *22*. This has the interesting property that *there is no distinction between binding*

[1] The use of λ for this purpose is historical; = or even $\Rightarrow$ would be more appropriate for BAWL.

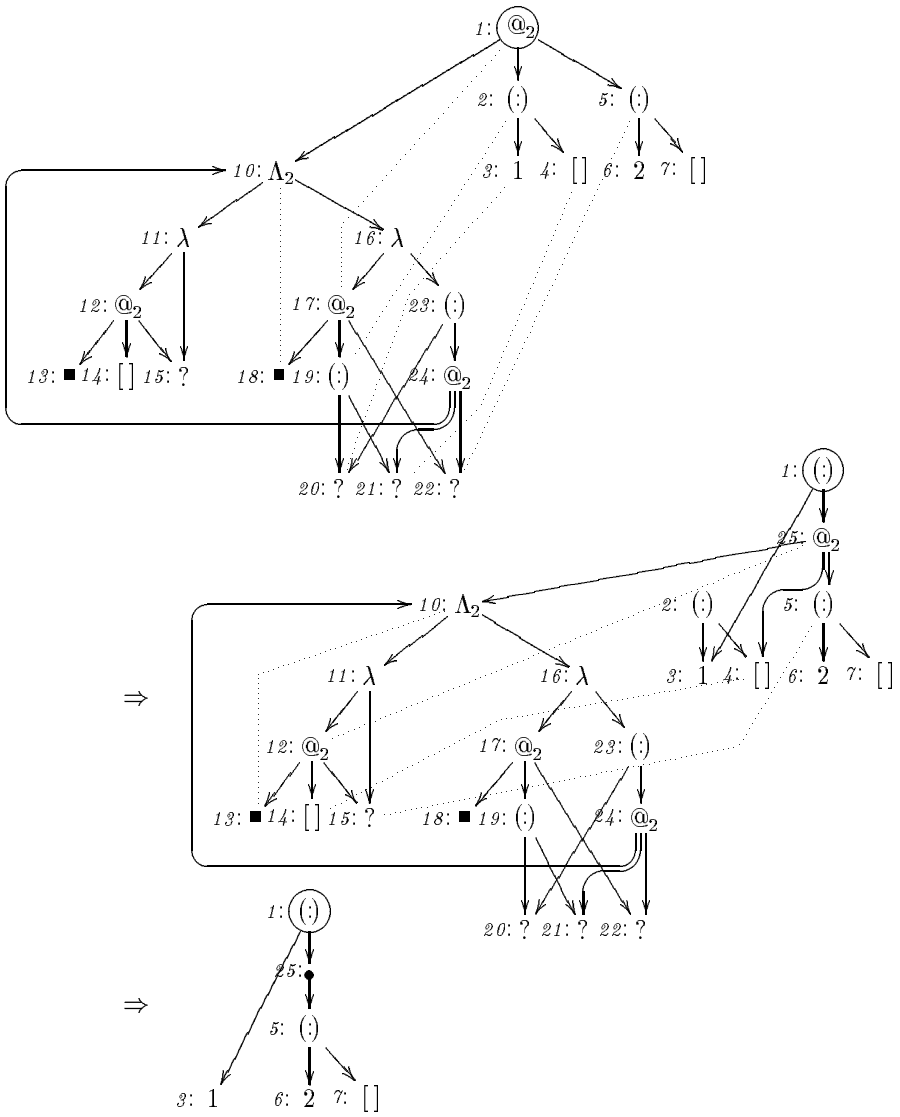

Figure 22.3 Term graph representation, matching, and reduction of BAWL expression

and other forms of acces to values. This is a crucial property that allows a much more uniform definition of suspensions, sharing, etc. as we shall see, by eliminating the need for auxiliary structure like environments and stores.

We want to reduce the application—so we compare the two LHS subgraphs (at *12* and *17*) with the application itself (at *1*). It is clear that (only) the LHS of the right equation fits: we match *17* to *1*, *18* to *10*, *19* to *2*, *20* to *3*, *21* to *4*, and *22* to *5* as shown by the dotted lines in the top graph; in the original program this corresponds

to binding $x = 1$ (by mapping *20* to *3*), $xs = [\,]$ (by mapping *21* to *4*), and $ys = [2]$ (by mapping *22* to *5*). The ■ is convenient because it makes the comparison simpler.

Reducing an application is the same as replacing it with the body of the function (in this case the RHS at *23*) after substitution of the bound variables with whatever they were bound to. This is accomplished with the middle graph in figure 22.3 where we see how node *1* has been overwritten by a copy of node *23* that points to a fresh node *25*—we have *build new structure* on top of the arguments corresponding to the nodes unique to the RHS; the actual arguments remain shared. Notice how the (:)-node at *2* is no longer referenced by this expression—if it is not referenced from other parts of the graph then it has become garbage to be collected for reuse.

Anyway, there is now a new redex[2] in the graph as indicated by the dotted lines in the middle graph. There is one complication in this case, however: there are no unique nodes in the RHS and thus nothing to overwrite with! One way to solve this is by introducing *indirection nodes*, and this is what we have done in the reduced bottom graph of figure 22.3: node *25* has been overwritten by an indirection (the •) to the corresponding list argument (at *5*) that may thus remain shared. We have taken the liberty of removing the garbage (which includes the entire representation of the ++ function as well as node *2* and *4*) from this last graph.

It is clear that the above reduction captures many aspects of evaluation of lazy functional languages that are difficult to express in most other formalisms. This is the primary motivation for GOS and the subject of this chapter.

Background and related work

This chapter grew out of the desire in the author's thesis [Ros92b] to describe the semantics of graph reduction in lazy functional languages using graphs—even though this is an obvious idea most authors seem to prefer using stores, environment frames, substitutions, etc.

However, the idea has been around since [Wad71] who used it to implement the λ-calculus with acyclic sharing of common subterms. Modeling recursion by cyclic sharing was suggested for data by [Hoa75] and for functions by [Hug82]. This is also where the intuition of why graph reduction provides a useful model for lazy evaluation of functional languages is presented—"just keep all the subexpressions around, updating each to its proper value only as needed"—although it had to some extent been folklore since [McC60] and has been used extensively in implementations of functional languages. Examples are the Functional Abstract Machine [Car83], the G-machine [Joh84, BPJR88], Clean [BvEvLP87, KSvEP91], and DACTL [Ken88, GKS89]. All of these can be perceived as notations for algorithms that describe how graphs may be used to model the evaluation of the implemented language. Such notations have the advantage that they are directly executable. However, they are often not very readable since all the details of the implementation have to be present in the specification.

On the other hand, the idea has also been used in attempts to build models of functional languages and to prove properties of such models directly. Again starting from [Wad71] such attempts are naturally focused on models of the λ-calculus—a

[2] We will use the traditional word "redex" (reducible expression) rather than some abbreviation of "reducible subgraph" even though we will always refer to the latter.

rather large are of research, so we will just mention [Sta78] and [Lam90] as inspirational sources.

Overview

In section 22.2 we summarize the necessary TGR concepts in a notation convenient for our use, and we present the "minimal lazy rewrite" theorem that will prove useful later. We then specify the graph representation of BAWL programs in section 22.3 and present GOS by giving the dynamic semantics of BAWL in section 22.4—these two sections together outline a fully lazy operational semantics for BAWL. Finally we briefly discuss applications of GOS.

22.2 TERM GRAPH REWRITING

We briefly summarize term graphs and the notation of this chapter, impose a *match order* on them and define *rewriting* from this. The definitions in this chapter are based on those of [BvEG+87]; readers familiar with that paper (or other chapters in this book) will not find many surprises in this section except perhaps for the declarative definition of rewriting that is a compromise between the axiomatic one used by categorical treatments and the operational one used elsewhere.

We will make use of basic notation for sets, functions, relations, and orderings. In particular: Tuples satisfy $\langle x_1, \ldots, x_k \rangle.i = x_i$. A function $\phi\colon X \to Y$ has domain $\mathrm{Dm}(\phi)$ and range $\mathrm{Rg}(\phi)$; maps may be written $\{x \mapsto y, \ldots\}$ or even $\{\, x \mapsto y \mid p(x,y) \,\}$; restriction is written $\phi|_X$, and update $\phi[\psi]$ is ϕ updated with the mappings of ψ (so the mappings $\{\, x \mapsto \phi(x) \mid x \in \mathrm{Dm}(\phi) \cap \mathrm{Dm}(\psi) \}$ disappear). N is the set of natural numbers (including 0), and $\mathcal{I}$ is the set of identifiers.

DEFINITION **22.2.1** *Given a set of "labels" $\mathcal{L}$ with an associated "arity" function $\#\colon \mathcal{L} \to \mathsf{N}$ and a set of "nodeids" $\mathcal{N} = \{1, 2, \ldots\}$. A* term graph $\mathbf{g} \in \mathcal{G}_{\mathcal{L},\#}$ *is a "rooted labeled ordered directed graph", i.e., a structure $(N_\mathbf{g},\ lab_\mathbf{g},\ succ_\mathbf{g},\ r_\mathbf{g})$ where*

$N_\mathbf{g}$	$\subset$	$\mathcal{N}$	the *nodeids*
$lab_\mathbf{g}$	:	$N_\mathbf{g} \to \mathcal{L}$	maps each node to its *label*
$succ_\mathbf{g}$	:	$N_\mathbf{g} \to N_\mathbf{g}^*$	maps each node to its *successor tuple*
$r_\mathbf{g}$	$\in$	$N_\mathbf{g}$	is the *root node*

that respects the label arities, i.e., $\forall n \in N_\mathbf{g} : succ_\mathbf{g} n = \langle n_1, \ldots, n_{\#(lab_\mathbf{g} n)} \rangle$. We use the abbreviation $\ell_\mathbf{g} = lab_\mathbf{g} r_\mathbf{g}$.

Note that (term) graphs are not required to be connected—we do not do implicit garbage collection as [BvEG+87] but as shown in that paper this is of no consequence. Graphs always have a connected part containing at least the root node, however: there is no "null graph". Also we only allow one root node in each graph; this can be circumvented by using a dummy root node where only its successors are of interest.

We manipulate (term) graphs using the following operations and concepts, although we will prefer to draw them.

DEFINITION **22.2.2** *a. The* ith successor graph *is* $\mathbf{g}.i = (N_\mathbf{g}, lab_\mathbf{g}, succ_\mathbf{g}, (succ_\mathbf{g} r_\mathbf{g}).i)$.

b. The reachable nodes *are defined by* $reach(\mathbf{g}) = reach'\langle \mathbf{g}, \emptyset \rangle$ *where*

$$\begin{aligned} reach'\langle \mathbf{g}, N \rangle &= \begin{cases} N & \text{if } r_{\mathbf{g}} \in N \\ N_1 \cup \cdots \cup N_k & \text{otherwise} \end{cases} \\ k &= \#\ell_{\mathbf{g}} \\ N_i &= reach'\langle \mathbf{g}.i, N \cup \{r_{\mathbf{g}}\} \rangle \quad \text{for} \quad 1 \le i \le k \end{aligned}$$

c. The garbage collection *of a graph contains only the reachable nodes:* $gc(\mathbf{g}) = (N, lab_{\mathbf{g}}|_N, succ_{\mathbf{g}}|_N, r_{\mathbf{g}})$, *where* $N = reach(\mathbf{g})$. $\mathbf{g}$ *is* connected *iff* $\mathbf{g} = gc(\mathbf{g})$.

d. A (proper) subgraph $\mathbf{g}|n = gc(N, lab_{\mathbf{g}}, succ_{\mathbf{g}}, n)$ *for* $n \in reach(\mathbf{g})$.

e. Rerooting *a graph means "change the root nodeid to something else and 'redirect' all arrows from the old to the new root", defined by* $r{:}\,\mathbf{g} = (N, lab, succ, r)$ *where*

$$\begin{aligned} N &= \{r\} \cup (N_{\mathbf{g}} \setminus \{r_{\mathbf{g}}\}) \\ lab &= lab|_{N_{\mathbf{g}} \setminus \{r_{\mathbf{g}}\}}[r \mapsto \ell_{\mathbf{g}}] \\ (succ\, n).i &= \begin{cases} r & \text{if } n_i = r_{\mathbf{g}} \\ r_{\mathbf{g}.i} & \text{if } n_i \neq r_{\mathbf{g}} \wedge n = r \\ n_i & \text{if } n_i \neq r_{\mathbf{g}} \wedge n \neq r \end{cases} \\ & \quad \text{where} \quad n_i = (succ_{\mathbf{g}} n).i \end{aligned}$$

f. An insertion $\mathbf{g}[\mathbf{x}] = \big(N_{\mathbf{g}} \cup N_{\mathbf{x}},\ lab_{\mathbf{g}}[lab_{\mathbf{x}}],\ succ_{\mathbf{g}}[succ_{\mathbf{x}}],\ r_{\mathbf{g}}\big)$ *is a way to replace parts of a graph* $\mathbf{g}$ *with the parts of* $\mathbf{x}$ *that have nodeids found in* $N_{\mathbf{g}}$.

g. A term graph set $\mathcal{G}_{\mathcal{L},\#}$ *is* arity monotonic *iff for all* $\ell, \ell' \in \mathcal{L}$ *we have that* $\ell \sqsubseteq \ell'$ *implies* $\#\ell \le \#\ell'$.

Choosing rerooting and insertion instead of the traditional "redirection" and "context" means that these operations are very close to operations available on graph reduction machines—and in fact the drawings we use can be represented as a string of insertions (constructing the "dominant graph") and rerootings (the "backpointers").

With this we are ready to define and use graph matching and rewriting.

DEFINITION **22.2.3** *Given* $\mathbf{p}, \mathbf{a} \in \mathcal{G}_{\mathcal{L},\#}$. $\sigma : N_{\mathbf{p}} \to N_{\mathbf{a}}$ *is a* rooted $\sqsubseteq$-match map from $\mathbf{p}$ to $\mathbf{a}$, *written* $\sigma \triangleright \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a}$, *iff it satisfies* $\forall n \in reach(\mathbf{p})$*:*

$$lab_{\mathbf{p}}(n) \sqsubseteq lab_{\mathbf{a}}(\sigma(n)) \tag{22.1}$$

$$\sigma(succ_{\mathbf{p}}(n).i) = succ_{\mathbf{a}}(\sigma(n)).i \qquad \forall i \in \{1, \ldots, \#(lab_{\mathbf{p}}(n))\} \tag{22.2}$$

This is a generalization of the usual graph substitution concept in that it is parameterised over $\sqsubseteq$. In fact, matching with a *discrete* ordering corresponds to the "rooted homomorphism"—symbolically $\xrightarrow{=}$ is $\to$ of [BvEG+87]. Its symmetric closure is the "rooted isomorphism" ($\approx$) equivalence relation. Similarly, matching with a *flat* ordering (like $\xrightarrow{=_\perp}$) corresponds to the rooted homomorphism on "open graphs" of [BvEG+87] except that we include the "empty" $\perp$ nodes in the graph as actual nodes rather than leave them out. We discuss in [Hol90b] how our parametrization can be generalized to provide encoding of type systems.

It is easy to see that such a match is always unique (proof in [Ros92b]).

THEOREM **22.2.4** *Given* $\mathbf{p}, \mathbf{a} \in \mathcal{G}_{\mathcal{L},\#}$, $\mathcal{G}_{\mathcal{L},\#}$ *arity-monotonic. Then* $\sigma \triangleright \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a}$ *iff*

$$\begin{array}{rcl} \sigma(r_{\mathbf{p}}) = r_{\mathbf{a}} & \wedge & \ell_{\mathbf{p}} \sqsubseteq \ell_{\mathbf{a}} \\ & \wedge & \forall i \in \{1, \ldots, \#\ell_{\mathbf{p}}\} : \sigma' \triangleright \mathbf{p}'|_{r_{\mathbf{p}}.i} \xrightarrow{\sqsubseteq} \mathbf{g}.i \\ & & \quad \text{where} \quad \mathbf{p}' = \mathbf{p}[r_{\mathbf{p}} : \ell_{\mathbf{p}}\langle\rangle],\ \sigma' = \sigma[\mathbf{p}' \mapsto \mathbf{g}] \end{array}$$

This theorem is the key to rewriting by the "extending the match" idea used in the categorical models (probably described in other chapters of this book):

DEFINITION **22.2.5** *Given* $\mathbf{p}, \mathbf{a}, \mathbf{b} \in \mathcal{G}_{\mathcal{L},\#}$ *such that* $\sigma \triangleright \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a}$. *Then* σ describes the term graph rewrite of $\mathbf{b}$ to $\mathbf{c}$, *written* $\sigma \triangleright \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a} \triangleright \mathbf{b} \xrightarrow{\sqsubseteq} \mathbf{c}$, *iff*

$$\exists \sigma' : \sigma'|_{N_{\mathbf{p}}} = \sigma \quad \wedge \quad \sigma' \triangleright \mathbf{b} \xrightarrow{\sqsubseteq} \mathbf{c}$$

And in fact the constructive proof of Theorem 22.2.4 is an algorithm that produces the minimal contractum $\mathbf{c}$.

COROLLARY **22.2.6** *Given arity-monotonic* $\mathcal{G}_{\mathcal{L},\#}$ *with* $\sigma \triangleright \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a}$. *Then* $\mathbf{c}_m$ *given by*

$$\begin{array}{l} \mathbf{c}_m = \sigma_m(gc(\mathbf{b})) \\ \sigma_m = \mathit{rewrite}\ \{\, n \mapsto n \mid n \in \mathit{reach}(\mathbf{b}) \setminus \mathit{reach}(\mathbf{p}) \,\}\ \mathbf{b} \\ \qquad \textbf{where} \quad \mathit{rewrite}\ \sigma\ \mathbf{b} = \sigma, \quad \text{if } \mathbf{b} \in \mathit{Dm}(\sigma) \\ \qquad\qquad\qquad\qquad\quad = \sigma_k, \text{ if } \mathbf{b} \notin \mathit{Dm}(\sigma) \\ \qquad\qquad\qquad\qquad\quad\ \textbf{where} \quad k = \#\ell_{\mathbf{b}} \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ c \ \text{ is a fresh nodeid} \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ \sigma_0 = \sigma[r_{\mathbf{b}} \mapsto c] \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ \sigma_i = \mathit{rewrite}\ \sigma_{i-1}\ \mathbf{b}.i \end{array}$$

satisfies $\forall \mathbf{c}, \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{a} \triangleright \mathbf{b} \xrightarrow{\sqsubseteq} \mathbf{c} : \mathbf{c}_m \xrightarrow{\sqsubseteq} \mathbf{c}$.

Essentially this is just the usual search for a "maximal glue component" but in the parameterised case; we will use this to specify fully lazy reduction later, but it may also be used for other things, e.g., when encoding a type system in $\sqsubseteq$ this can be used to find a most specific type.

The reader is invited to verify that $\{17 \mapsto 1,\ 18 \mapsto 10,\ 19 \mapsto 2,\ 20 \mapsto 3,\ 21 \mapsto 4,\ 22 \mapsto 5\} \triangleright \mathbf{g}|_{17} \xrightarrow{\sqsubseteq} \mathbf{g}|_1$ in the top graph of figure 22.3, provided the $\sqsubseteq$-ordering satisfies $\blacksquare \sqsubseteq \Lambda_2$, $? \sqsubseteq []$, $? \sqsubseteq (:)$, $? \sqsubseteq 1$, and $? \sqsubseteq 2$.

22.3 GRAPH REPRESENTATION OF BAWL PROGRAMS

This section discusses and defines the term graph representation of BAWL programs. We first make a point of giving a perfectly naïve term graph representation and then we argue that the kind of matching and rewriting we wish to describe can be performed easily on this representation by imposing a suitable label ordering on it. Finally we comment on the how to interpret graph values as BAWL values.

We define the representation of (Core) BAWL programs as graphs in figure 22.4: the $\rho \vdash_X$ prefix should be read "the bindings $\{I \mapsto \mathbf{g}\}$ in ρ makes the following a well-formed X", where X is a syntactic BAWL component (of Definition 22.1.1) as shown. If

$$\frac{\rho[\rho'] \vdash_S \; S \qquad \rho[\rho'] \vdash_E \; E \rightsquigarrow \mathbf{e}}{\rho \vdash_B \; \begin{array}{l} S \\ ?\ E \end{array} \rightsquigarrow \mathbf{e}} \tag{22.3}$$

$$\frac{\rho \vdash_D \; D_1 \rightsquigarrow \mathbf{d}_1 \quad \cdots \quad \rho \vdash_D \; D_k \rightsquigarrow \mathbf{d}_k}{\rho \vdash_S \; \begin{array}{l} D_1 \\ \vdots \\ D_k \end{array}} \tag{22.4}$$

$$\frac{\rho_i \vdash_P \; P_{ij} \rightsquigarrow \mathbf{p}_{ij} \qquad \rho[\rho_i] \vdash_S \; S_i \qquad \rho[\rho_i] \vdash_E \; E_i \rightsquigarrow \mathbf{e}_i}{\rho \vdash_D \; \begin{array}{l} I\ P_{11} \ldots P_{1k} \;=\; E_1 \textbf{ where } S_1 \\ \vdots \\ I\ P_{n1} \ldots P_{nk} \;=\; E_n \textbf{ where } S_n \end{array} \rightsquigarrow \begin{array}{c} \Lambda_n \\ \lambda \quad \cdots \quad \lambda \\ @_k \quad \mathbf{e}_1 \quad @_k \quad \mathbf{e}_n \\ \blacksquare\ \mathbf{p}_{11} \cdots \mathbf{p}_{1k} \quad \blacksquare\ \mathbf{p}_{n1} \cdots \mathbf{p}_{nk} \end{array}} \; \forall ij \tag{22.5}$$

$$\frac{}{\rho \vdash_P \; I \rightsquigarrow \; r{:}\,(?)} \; \rho(I) = r{:}\,(?) \qquad \frac{}{\rho \vdash_P \; !I \rightsquigarrow \; r{:}\,(!)} \; \rho(I) = r{:}\,(!) \tag{22.6}$$

$$\frac{\rho \vdash_P \; P_i \rightsquigarrow \mathbf{p}_i}{\rho \vdash_P \; (X\ P_1 \ldots P_k) \rightsquigarrow \begin{array}{c} r{:}\,(\ell) \\ \mathbf{p}_1 \quad \cdots \quad \mathbf{p}_k \end{array}} \; \forall i; \begin{cases} \ell = !!_k & \text{if } X = !! \\ \ell = C & \text{if } X = C \end{cases} \tag{22.7}$$

$$\frac{}{\rho \vdash_E \; I \rightsquigarrow \mathbf{e}} \; \rho(I) = \mathbf{e} \tag{22.8}$$

$$\frac{\rho \vdash_E \; E_i \rightsquigarrow \mathbf{e}_i}{\rho \vdash_E \; (\ C\ E_1 \ldots E_k\ ) \rightsquigarrow \begin{array}{c} r{:}\,(C) \\ \mathbf{e}_1 \quad \cdots \quad \mathbf{e}_k \end{array}} \; \forall i \tag{22.9}$$

$$\frac{\rho \vdash_E \; E_i \rightsquigarrow \mathbf{e}_i}{\rho \vdash_E \; (\ E_0\ E_1 \ldots E_k\ ) \rightsquigarrow \begin{array}{c} r{:}\,(@_k) \\ \mathbf{e}_0 \quad \cdots \quad \mathbf{e}_k \end{array}} \; \forall i \tag{22.10}$$

Figure 22.4 Graph representation of core BAWL

$\rightsquigarrow \mathbf{g}$ is specified then $\mathbf{g}$ is the corresponding graph representation. The representation is completely naïve, using a special label for each construction in Definition 22.1.1. The only trick used is the encoding of argument pattern abstractions using $@_i$ with a "dummy function plug" $\blacksquare$, i.e., such that they look similar to the encoding of the applications they should match as mentioned in the introduction. It is not difficult to show that $\rho, \mathbf{e}$ can be found for any BAWL program B using (22.3) that invokes the other rules as appropriate.

Now we are just left with designing the label domain such that the graph represen-

tation gives us the matching and rewriting that we wish to use. The chosen ordering is shown in figure 22.5 as a Hasse diagram. The triangles between $!!_k$ and $\mathcal{C}_i$ just mean

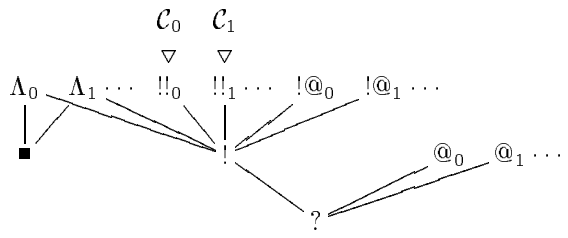

Figure 22.5 BAWL label ordering

that $!!_k$ should be the only label lesser than the collection of constructors of arity k.

The only thing that is complicated in this is the recognition of partial applications since partial and complete applications are not distinguishable syntactically, and yet only partial ones are in WHNF. As discussed in the following section we do it by dynamically "marking" them as different by updating @-nodes at the root of partial applications to !@-nodes that should be in WHNF, i.e., match !-nodes. The added $!@_k$-nodes in the figure clearly satisfy this. Note that $!@_k$ do not represent anything according to the rules of figure 22.4 since the *syntax* does not distinguish between complete and incomplete applications.

The design means that we can make maximal use of graph rewriting in the following section, e.g., both matches indicated by dotted lines in figure 22.3 are handled by this.

We have not yet considered how to interpret a result graph as a BAWL value. But fortunately this is not complicated: the rules of figure 22.4 may be used "backwards" without any problems as long as we create variable names and change any !@-nodes into @-nodes as mentioned above. There is even a smallest BAWL program representing the value of any graph thanks to Corollary 22.2.6.

22.4 GOS—GRAPH OPERATIONAL SEMANTICS

This section gives the operational semantics of BAWL by specifying a reduction relation on the BAWL graph representation of the previous section. We have chosen to base GOS on "Structural Operational Semantics" style of [Plo81, Kah87], since the generality of relations make them well-suited for definitions over graphs.

We start by discussing the goal of the specification, in particular how it may be separated into a module specifying the evaluation strategy and a module specifying the evaluation step. The two main modules, pattern driven evaluation and the evaluation step, are then discussed separately. Finally, we discuss a technical point: indirections.

The purpose of this entire exercise is to achieve a semantic description of BAWL that captures and communicates those aspects of the language that we wish to emphasize on and experiment with. The following are the main features:

- *Lazy evaluation*: only structure that is needed by the user should be evaluated.

- *Pattern driven reduction strategy*: the pattern matching should be used to determine the evaluation order.
- *Fully lazy curried applications*: the individual evaluation "steps" are applications of curried functions to arguments with minimal duplication of data achieved by *destructive updating* whenever possible.

Fortunately the first point is a special case of the second provided the standard "driver function" *seq* (mentioned in the introduction) is available, so we will not discuss that any further. The other two will be described by the following relations:

- $\mathbf{p} : \mathbf{e} \Downarrow \mathbf{g}$: "The pattern graph $\mathbf{p}$ drives the converging of the expression graph $\mathbf{e}$ to the value graph $\mathbf{g}$" (figure 22.6).
- $\mathbf{g} \Rightarrow \mathbf{g}'$: "The value graph $\mathbf{g}$ reduces in a single step to the value graph $\mathbf{g}'$" (figure 22.7).

$$\boxed{\mathbf{p} : \mathbf{e} \Downarrow \mathbf{g}}$$

$$\frac{}{\mathbf{p} : \mathbf{g} \Downarrow \mathbf{g}} \qquad \mathbf{p} \xrightarrow{\sqsubseteq} \mathbf{g} \tag{22.11}$$

$$\frac{\mathbf{p}_1 : \mathbf{g}_1 \Downarrow \mathbf{g}'_i \quad \cdots \quad \mathbf{p}_k : \mathbf{g}_k \Downarrow \mathbf{g}'_k}{p{:}\,(\ell)\ \mathbf{p}_1 \cdots \mathbf{p}_2 \ :\ g{:}\,(\ell)\ \mathbf{g}_1 \cdots \mathbf{g}_2 \ \Downarrow\ g{:}\,(\ell)\ \mathbf{g}'_1 \cdots \mathbf{g}'_2} \qquad \ell \in \{@, !@\} \cup \mathcal{C} \tag{22.12}$$

$$\frac{\mathbf{g} \Rightarrow \mathbf{g}' \qquad \mathbf{p} : \mathbf{g}' \Downarrow \mathbf{g}''}{\mathbf{p} : \mathbf{g} \Downarrow \mathbf{g}''} \tag{22.13}$$

Figure 22.6 The BAWL reduction strategy

Rule (22.11) is there to allow us to stop reduction when the pattern is "satisfied"—this requires a match and thus uses $\xrightarrow{\sqsubseteq}$ based on the label order from above. Rule (22.12) allows reduction of subgraphs driven by the corresponding subpatterns when the root labels are equal; the use of g on both sides of $\Downarrow$ ensures *destructive updating*, i.e., that all parts of the graph that refer to this constructed node will share the results of updating. Finally (22.13) is the rule for application and makes use of the "single step reduction" relation $\Rightarrow$ defined below. Notice that performing the reduction of an application does not depend on the pattern that drives the reduction. It is interesting to realize how simple it is to prove that this rule set uses destructive updating by induction over the possible proofs—the above three rules clearly satisfy this provided the $\mathbf{g} \Rightarrow \mathbf{g}'$ rules also do it!

Now for the evaluation step rules: (22.14) is the generic β-reduction rule where we reduce an application with at least the right number of arguments: First we let the pattern list (BAWL LHS) drive evaluation of the argument list until there is a match (the $\exists i$ is there to remind us that we have to search for the right i). Then we use the just established match to rewrite the corresponding body (BAWL RHS) to the contractum that should be the result of the entire operation! The rule as shown is not specified fully lazy—this is abstracted out—but it is easy to do this by explicitly inserting

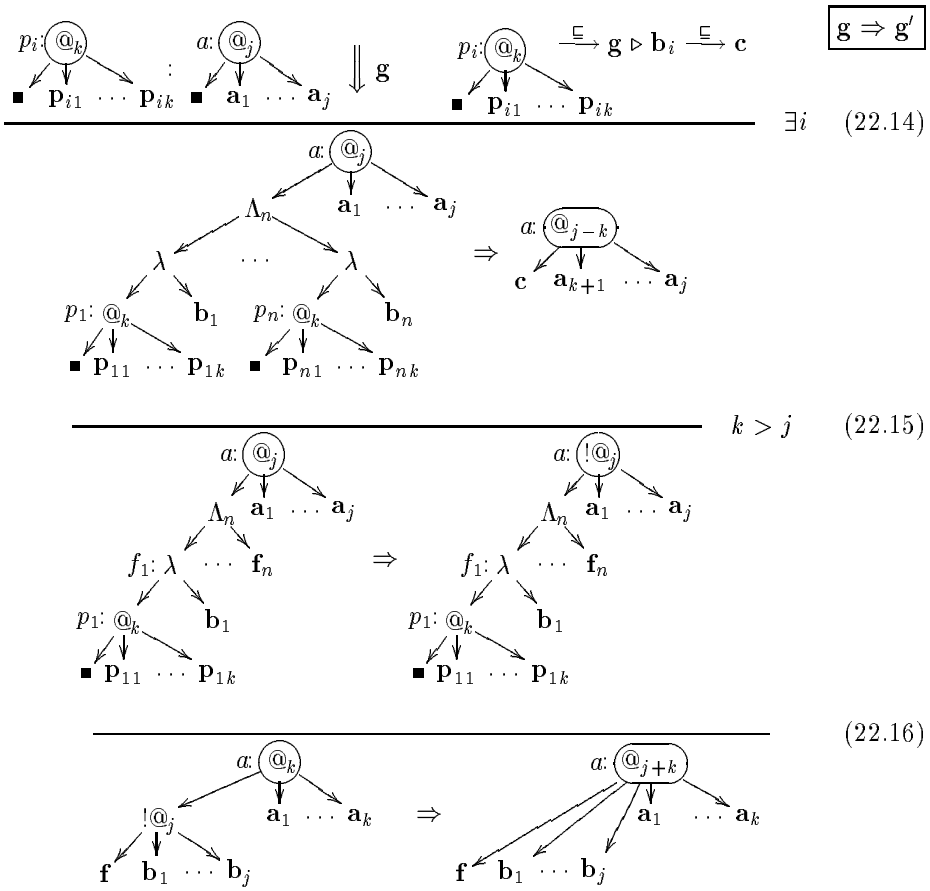

Figure 22.7 The BAWL evaluation step

the result of Corollary 22.2.6. (22.15) just updates an incomplete application when it is needed and (22.16) combines an incomplete application with the next outermost application; notice that this does not necessarily remove the partial application node itself.

Finally the promised discussion of *indirections*; fortunately this is easy. The thing to notice is that both (22.14) and (22.16) may succeed when $k = j$ in which case they leave a $@_0$-node as the root node. This node will either immediately merge with an underlying @-node by (22.16) or be updated to a $!@_0$-node by (22.15) that can only disappear in the same way. And a $!@_0$-node is indistinguishable from an indirection node; it even pattern matches as what it "points to"! The reader is invited to verify the example of figure 22.3 with the knowledge that • is in fact $@_0$ and $!@_0$.

There is only one situation where a $@_0$-node does not act as an indirection but instead as a *suspension*, and that is when it points to a nullary function. Then the entire thing is a redex that can be reduced immediately by (22.14), thus a suspension. But it is important to realize that such a suspension will never be generated and then

left alone by the evaluator since if it was generated then we still need to reduce it to get a match because it must come from another @-node that did not match!

However, there is a different, related issue: if the programmer specifies a BAWL *data definition* then this will be interpreted as a nullary function and consequently not be updated properly (instead each nullary "application" of the data will be updated). This is easily fixed, however, by adding a special case to the representation equations for data definitions such that the data is accessed directly (cf. [Ros92b, Ros91]).

22.5 DISCUSSION

We have sketched a specification of a definitely non-trivial lazy functional programming language, BAWL. The style of the presentation is of sufficient generality to express and manipulate the sharing properties of "real" programming languages in a direct way. We feel that this is a very promising direction that deserves further study, a study which we are currently undertaking.

Our approach as described here has been very "top-down": we started this by considering the needs of high-level lazy functional languages. Nevertheless, we have done some experiments with *implementation* of the presented ideas [Hol90a] in the form of a graph reduction machine called MatchBoxes, and particularly coding the efficient match and rewrite algorithm as primitives in the machine seems to have some promise.

Finally, this author's attempt at a "theoretical bottom-up" approach starting from first principles [Ros92a] is still fairly fresh. Yet we hope that the two may meet.

REFERENCES

[BPJR88] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *1988 ACM Conference on LISP and Functional Programming*, pp. 244–258, Snowbird, Utah, July 1988, ACM.

[BvEG+87] H. P. Barendregt, M. C. D. J. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven (editors), *PARLE '87—Parallel Architectures and Languages Europe vol. II*, vol. 256 of *LNCS*, pp. 141–158, Eindhoven, The Netherlands, June 1987, Springer-Verlag.

[BvEvLP87] T. Brus, M. C. D. J. van Eekelen, M. van Leer, and M. J. Plasmeijer. Clean—a language for functional graph rewriting. In G. Kahn (editor), *FPCA '87—Functional Programming Languages and Computer Architecture*, vol. 274 of *LNCS*, pp. 364–387, Portland, Oregon, 1987, Springer-Verlag.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Car83] L. Cardelli. *The Functional Abstract Machine.* Technical Report TR-107, Bell Labs, 1983.

[GKS89] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep. *Final Specification of Dactl.* Report SYS–C88–11, University of East Anglia, Norwich, UK, 1989.

[Hoa75] C. A. R. Hoare. Recursive data structures. *Journal of Computer and Information Sciences*, **4**, pp. 105–132, 1975.

[Hol90a] K. H. Holm. *Graph Matching in Functional Language Specification and Implementation.* skriftlig rapport 90-1-3, DIKU, Universitetsparken 1, DK-2100 København Ø, Denmark, December 1990.

[Hol90b] K. H. Holm. Graph matching in operational semantics and typing. In A. Arnold (editor), *CAAP '90—15th Colloqvium on Trees and Algebra in Programming*, vol. 431 of *LNCS*, pp. 191–205, Copenhagen, Denmark, March 1990, Springer-Verlag.

[Hug82] J. M. Hughes. Super-combinators. In *1982 ACM Symposium on LISP and Functional Programming*, pp. 1–10, Pittsburgh, Pennsylvania, August 1982, ACM.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. *Sigplan Notices*, **19**, 6, pp. 58–69, June 1984.

[Kah87] G. Kahn. *Natural Semantics.* Rapport 601, INRIA, Sophia-Antipolis, France, February 1987.

[Ken88] J. R. Kennaway. Implementing term rewrite languages in Dactl. In M. Dauchet and M. Nivat (editors), *CAAP '88—13th Colloquium on Trees in Algebra and Programming*, vol. 299 of *LNCS*, pp. 102–116, Nancy, France, March 1988, Springer-Verlag.

[KSvEP91] P. W. M. Koopman, S. E. W. Smetsers, M. C. D. J. van Eekelen, and M. J. Plasmeijer. Efficient graph rewriting using the annotated functional strategy. In Plasmeijer and Sleep [PS91], pp. 225–250. (available as Nijmegen Tech. Report 91-25).

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90—Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 16–30, San Francisco, California, January 1990, ACM.

[McC60] J. McCarthy. Recursive functions of symbolic expressions. *Communications of the ACM*, **3**, 4, pp. 184–195, April 1960.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo81] G. D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report FN-19, DAIMI, Aarhus University, Aarhus, Denmark, 1981.

[PS91] M. J. Plasmeijer and M. R. Sleep (editors). *SemaGraph '91 Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting*, Nijmegen, Holland, December 1991, Katholieke Universiteit Nijmegen. (available as Nijmegen Tech. Report 91-25).

[Ros91] K. H. Rose. Graph-based operational semantics for lazy functional languages. In Plasmeijer and Sleep [PS91], pp. 203–225. (available as Nijmegen Tech. Report 91-25).

[Ros92a] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy (editors), *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, LNCS, Pont-a-Mousson, France, July 1992, Springer-Verlag. Also available as DIKU semantics note D–143.

[Ros92b] K. H. Rose. *GOS—Graph Operational Semantics.* Speciale 92-1-9, DIKU, Universitetsparken 1, DK–2100 København Ø, Denmark, March 1992. (56pp).

[Sta78] J. Staples. A Graph-like Lambda Calculus for which Leftmost Outermost Reduction is Optimal. In V. Claus, H. Ehrig, and G. Rozenberg (editors), *1978 International Workshop in Graph Grammars and their Application to Computer Science and Biology*, number 73 in LNCS, pp. 440–454, Bad Honnef, F. R. Germany, 1978, Springer-Verlag.

[Tur82] D. A. Turner. Recursion Equations as a Programming Language. In J. Darlington, P. Henderson, and David A. Turner (editors), *Functional Programming and its Applications, an Advanced Course*, pp. 1–28. Cambridge University Press, 1982.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Programming Research Group, Oxford University, 1971.

Contents

Term Graph Rewriting: Theory and Practice, eds. M.R.Sleep, M.J.Plasmeijer and M.C. van Eekelen

List of Figures

Term Graph Rewriting: Theory and Practice, eds. M.R.Sleep, M.J.Plasmeijer and M.C. van Eekelen