Proc. TAPSOFT '91, LNCS 494, Springer Verlag 1991, pp. 438–457

From Reduction Machines To Narrowing Machines

Rita Loogen
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

Abstract

Narrowing, the evaluation mechanism of functional logic languages, can be seen as a generalization of *reduction*, the evaluation mechanism of purely functional languages. The unidirectional pattern matching, which is used for parameter passing in functional languages, is simply replaced by the bidirectional *unification* known from logic programming languages. We show in this paper, how to extend a reduction machine, that has been designed for the evaluation of purely functional programs to a machine that performs narrowing. The necessary extensions concern the realization of unification and backtracking. The latter has to be incorporated to handle nondeterministic computations. It turns out that the resulting narrowing machine can also be seen as an extension of Warren's Prolog engine [Warren 83]. This extension enables a space efficient handling of nested expressions and embodies an optimized treatment of deterministic computations. As in Warren's machine the central component of the machine is a stack that contains environments, i.e. activation records of function calls, and choice points to keep track of possible alternative computations. It is ensured that choice points always contain the minimal amount of information that is necessary to restore a previous state on backtracking. A complete specification of the machine and of the translation of a sample language into abstract machine code is given. To test the feasibility of the new implementation technique a preliminary implementation has been developed in Miranda.

1 Introduction

The integration of the functional and logic programming paradigms has been extensively investigated during the last years (for surveys see e.g. [DeGroot, Lindstrom 86] and [Bellia, Levi 86]). Logic languages have more expressive power than functional languages while the latter have a simpler execution model based on the *reduction principle*. Reduction applies when an expression matches the left-hand side of a program statement (function definition) and consists in replacing the expression by the corresponding right-hand side.

Functional logic languages are extensions of functional languages with principles derived from logic programming. While their syntax almost looks like the syntax of conventional functional languages, their operational semantics is based on *narrowing*, an evaluation mechanism that uses unification instead of pattern matching for parameter passing. Narrowing is a natural extension of reduction to incorporate unification. It means applying the minimal substitution to an expression in order to make it reducible, and then to reduce it.

Reduction machines for compiler implementations of functional languages, in general, make use of a stack to control the reduction process. The stack contains frames representing the environment of function calls, e.g. the actual parameters and storage for the local variables of the function definition. Of course, higher-order functions and/or data structures additionally require the use of a heap or graph structure. A survey of different techniques can e.g. be found in [Field, Harrison 88].

Compiler implementations of logic programming languages usually are variations of *Warren's Prolog Engine* [Warren 83] whose main components are a stack, a heap and a trail. The stack is used to store both the environments (stack frames) of clauses and so-called choice or backtrack points indicating possible alternative computations and containing information necessary to restore previous states of the machine. The heap, which is organized as a stack, is used for the construction of lists and structures. The trail contains references to variables that have been bound during unification and that must be unbound on backtracking.

Taking these well-known techniques for the implementation of functional and logic languages as a basis we develop in this paper a technique for the sequential implementation of functional logic languages whose operational semantics is based on narrowing. On the one hand, the narrowing machine that will be presented is an extension of a stack-based reduction machine by components that are necessary for the realization of unification, the more general parameter passing mechanism, and backtracking which must be integrated to handle alternative computation paths induced by the generalized parameter passing. On the other hand, the machine can be seen as an extension of Warren's Prolog engine by features that enable the handling of nested expressions and lazy evaluation. Furthermore, a special property (nonambiguity) of the class of functional logic programs that we consider makes an optimized treatment of deterministic computations possible. Up to now the narrowing machine has been implemented in Miranda* to test its feasibility. A more elaborate implementation in C is in preparation.

The paper is organized as follows. In section 2 we describe the syntax and operational semantics of a sample functional logic language, called Simple BABEL. Section 3 presents the heart of the narrowing machine. The compilation of Simple BABEL programs into machine code is specified in section 4. In section 5 we report on important extensions that are necessary to implement higher-order functions and lazy evaluation. Section 6 describes the current state of the implementation. A discussion of related work is given in section 7. Section 8 finally contains some conclusions.

2 A Simple Functional Logic Language

In this section we define a small abstract language called *Simple BABEL*. It corresponds to a subset of the functional logic language BABEL [Moreno,Rodríguez 88,89]. Simple BABEL is a first-order weakly typed functional logic language based on a constructor discipline. Its operational semantics is innermost narrowing. The restriction to Simple BABEL simplifies the explanation and specification of the narrowing machine. The extensions that are necessary to cope with 'full' BABEL, will be discussed in section 5.

2.1 Syntax of Simple BABEL

Let $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ be ranked alphabets of *constructors* and *function symbols*, respectively. We assume the nullary constructors 'true' and 'false' to be predefined. Predefined function symbols are the boolean operators and the equality operator. In the following, letters $c, d, e \ldots$ are used for constructors and the letters $f, g, h \ldots$ for function symbols.

The following syntactic domains are distinguished:

- *Variables* $X, Y, Z \ldots \in Var$
- *Terms* $s, t, \ldots \in Term$: $\quad t ::= X \quad$ % Variable
  $\qquad\qquad\qquad\qquad\quad \mid c(t_1, \ldots, t_n) \quad$ % $c \in DC^n, n \geq 0$
- *Expressions* $M, N \ldots \in Exp$:

*Miranda is a trademark of Research Software Limited.

$$
\begin{array}{rcll}
M & ::= & t & \%\ t \in \mathit{Term} \\
& | & c(M_1, \ldots, M_n) & \%\ c \in DC^n, n \geq 0 \\
& | & f(M_1, \ldots, M_n) & \%\ f \in FS^n, n \geq 0 \\
& | & (B \to M) & \%\ \text{guarded expression} \\
& | & (B \to M_1 \Box M_2) & \%\ \text{conditional expression}
\end{array}
$$

$B \to M$ and $B \to M_1 \Box M_2$ are intended to mean "**if** B **then** M **else** not defined" and "**if** B **then** M_1 **else** M_2", respectively.

A *Simple BABEL-program* consists of a finite set of defining rules for the not predefined function symbols in FS. Let $f \in FS^m$. Each *defining rule for f* must have the form:

$$f(t_1, \ldots, t_m) := M$$

and satisfy the following conditions:

1. *Data Pattern*: $t_i \in \mathit{Term}$.
2. *Left Linearity*: $f(t_1, \ldots, t_m)$ does not contain multiple variable occurrences.
3. *Local determinism*: $vars(f(t_1, \ldots, t_m)) \supseteq vars(M)$.
4. *Nonambiguity*: Given any two rules for the same function symbol f:

   $$f(t_1, \ldots, t_m) := M \text{ and } f(s_1, \ldots, s_m) := N,$$

   if $f(t_1, \ldots, t_m)$ and $f(s_1, \ldots, s_m)$ have a most general unifier σ then $M\sigma, N\sigma$ are identical.

   $M\sigma$ denotes the expression M where all variables X have been replaced by $\sigma(X)$.

If necessary, the following rules for the predefined function symbols are implicitly added to a Simple BABEL program.

- Rules for the boolean operations:

$\neg$ false	:=	true
$\neg$ true	:=	false

false $\wedge$ Y	:=	false
true $\wedge$ Y	:=	Y

false $\vee$ Y	:=	Y
true $\vee$ Y	:=	true

- Rules for the equality operator '=':

$(c = c)$	:= true	$\%\ c \in DC^0$,
$(c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n))$	:= $(X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n)$	
		$\%\ c \in DC^n$, $n > 0$
$(c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_m))$	:= false	$\%\ c \in DC^n$, $d \in DC^m$, $c \neq d$ or $n \neq m$

2.2 Narrowing

Narrowing is described inductively. Narrowing rules define local computation steps. The narrowing relation specifies rewriting within expressions.

The evaluation mechanism of functional languages (pattern matching and reduction) only allows the evaluation of expressions which do not contain free variables. By replacing pattern matching by unification it is possible to evaluate general expressions. Unification allows to bind free variables to patterns given in the rules and thus to generate an instantiation of the given expression that is reducible. The result of a narrowing sequence is a modified expression and a substitution for the free variables in the original expression. In general several outcomes are possible for an expression containing free variables.

2.2.1 Narrowing Rules

$$\longrightarrow_\sigma \subseteq Exp \times Exp \text{ with } \sigma : Var \rightarrow Exp$$

Let ε denote the empty substitution, i.e. $\varepsilon(X) = X$ for all $X \in Var$.

1. $(\text{true} \rightarrow M) \longrightarrow_\varepsilon M$ $\quad$ $(\text{true} \rightarrow M_1 \Box M_2) \longrightarrow_\varepsilon M_1$ $\quad$ $(\text{false} \rightarrow M_1 \Box M_2) \longrightarrow_\varepsilon M_2$
2. Let $f(t_1, \ldots, t_m) := R$ be a rule of some fixed program and $f(M_1, \ldots, M_m)$ an expression. If $\theta \cup \sigma : Var \rightarrow Exp$ is a most general unifier with $t_i\theta = M_i\sigma$ for $1 \leq i \leq m$, then:

$$f(M_1, \ldots, M_m) \longrightarrow_\sigma R\theta.$$

The second narrowing rule describes the application of a Simple BABEL rule for the evaluation of a function application and corresponds to the copy rule or β-reduction in the reduction semantics of functional languages. Simple pattern matching corresponds to the determination of a substitution $\theta : Var \rightarrow Exp$ with $t_i\theta = M_i$, while *unification* of 'pattern' t_i with expressions M_i explicitly allows the binding of variables in M_i. Such bindings are given by the subfunction σ of the most general unifier $\theta \cup \sigma$. θ describes the binding of local variables in the BABEL rule.

2.2.2 Narrowing Relation

The *narrowing relation* $\Longrightarrow_\sigma \subseteq Exp \times Exp$ where $\sigma : Var \rightarrow Exp$ is inductively defined by:

- If $M_i \longrightarrow_\sigma N_i$ for $i \in \{1, \ldots, n\}$ then
  - $c(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma c(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
  - $f(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma f(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
- $B \longrightarrow_\sigma B'$ implies
  - $(B \rightarrow M) \Longrightarrow_\sigma (B' \rightarrow M\sigma)$ $\quad$ and $\quad$ $(B \rightarrow M_1 \Box M_2) \Longrightarrow_\sigma (B' \rightarrow M_1\sigma \Box M_2\sigma)$

The execution of several computation steps is given by the transitive, reflexive closure of the narrowing relation with composition of the substitutions, $\stackrel{*}{\Longrightarrow}_\sigma$.
Narrowing of a Simple BABEL expression M may lead to the following outcomes:

- *Success:* $M \stackrel{*}{\Longrightarrow}_\sigma t$ with $t \in Term$,
- *Failure:* $M \stackrel{*}{\Longrightarrow}_\sigma N$, N is not further narrowable and $N \notin Term$,
- *Nontermination.*

For simplicity we consider in the following sections first the *leftmost innermost* narrowing strategy.

2.3 Programming in Simple BABEL

A Simple BABEL program looks like a functional program. Additional expressive power is provided by the free (logic) variables that may be contained in the expressions that are to be evaluated. The definition of the append function for list concatenation

$$\begin{array}{lcl} append\,([\,], Z) & := & Z. \\ append\,([X \mid Y], Z) & := & [X \mid append(Y, Z)]. \end{array}$$

can e.g. be used

- in a purely functional way, as in $append([a,b],[b,a])$,
- to compute list differences, as in the expression

$$(append(X,[a,b]) = [b,a,a,b]) = \text{true}$$

- or to test, if a list is a sublist of another list:

$$(*) \quad (append(X, append([a,b],Z)) = [b,a,b,a,b]) = \text{true}.$$

Note that it is not advisible to use the expression $(append(X,[a,b]) = [b,a,a,b])$ to compute the difference of the two lists, as it would yield, in addition to the unique solution true, binding X to the list $[b,a]$, infinitely many successful computations with the result false, binding X to lists different from $[b,a]$.

A successful innermost narrowing sequence of the expression $(*)$ is the following:

$$\begin{array}{ll}
\multicolumn{2}{l}{(append(X, \underline{append([a,b],Z)}) = [b,a,b,a,b]) = \text{true}} \\
\overset{*}{\Longrightarrow}_{\varepsilon} & (\underline{append(X,[a \mid [b \mid Z]])} = [b,a,b,a,b]) = \text{true} \\
\Longrightarrow_{\{X/[X'|Y']\}} & (\underline{[X' \mid append(Y',[a \mid [b \mid Z]])] = [b,a,b,a,b]}) = \text{true} \\
\overset{*}{\Longrightarrow}_{\{X'/b\}} & (\underline{append(Y',[a \mid [b \mid Z]])} = [a,b,a,b]) = \text{true} \\
\Longrightarrow_{\{Y'/[]\}} & (\underline{[a \mid [b \mid Z]] = [a,b,a,b]}) = \text{true} \\
\overset{*}{\Longrightarrow}_{\{Z/[a,b]\}} & \text{true} = \text{true} \\
\Longrightarrow_{\varepsilon} & \text{true}
\end{array}$$

It yields the variable bindings $\{X/[b], Z/[a,b]\}$.

3 The Narrowing Machine

A *goal* for a given Simple BABEL program is any expression M. To solve a goal, the narrowing machine tries to reduce it to a normalized form. This means that the left hand sides of rules for the defined and predefined function symbols are unified with appropriate subexpressions, which are then replaced by the corresponding instances of the rule's right hand sides. The machine tries the program's rules in their textual ordering and evaluates arguments from left to right; it backtracks when a failure or a user's request for alternative solutions occurs.

3.1 Components of the Store

The store of the narrowing machine contains the following six components:

- *program store* $ps : PAdr \longrightarrow Instr$
  The program store contains the translation of the program rules into abstract machine code. This component remains unchanged during the evaluation of programs.

  We choose $PAdr := \mathbb{N}$. The set *Instr* of machine instructions will be explained later.

- *instruction pointer* $ip \in PAdr$
  The instruction pointer points at the address of the next instruction in the program store that has to be executed.

- *data stack* $d \in Adr^*$
  The data stack is used for all accesses to the graph, which contains the representations of terms. The data stack entries are graph (heap) addresses, $Adr := \mathbb{N}$.

  The arguments of function calls or constructor applications will be passed via the data stack and the result of these evaluations will be returned on top of this stack. Furthermore it is used during unification to organize the scanning of argument terms.

- *(environment) stack* $st \in (Adr \cup PAdr \cup \mathbb{N} \cup \{?\})^*$
  The environment stack is the central component of the machine. It is used to store the environment of function calls as in a reduction machine. For the realization of narrowing one furthermore needs to store special control information (choice points) in order to keep track of possible alternative computations. Thus, environments contain the control information for forward computations while choice points control backward computations, i.e. backtracking.

  The environment stack is accessed via two pointers:

  - the *environment pointer* $ep \in \mathbb{N}$ indicates the topmost environment (function or activation block) on the stack;
  - the *backtrack pointer* $bp \in \mathbb{N}$ indicates the topmost choice point.

  The *environments* of function calls have the same structure as in reduction machines, say

  $$\langle nlv, lvars, args, sep, ra \rangle$$

  where

  - $nlv \in \mathbb{N}$ is the number of storage locations reserved for local variables in this environment block.
  - $lvars \in (Adr \cup \{?\})^{nlv}$ are nlv stack positions for local variables. When the environment block is created, these positions are initialized with the symbol ? to indicate that a binding has not yet occurred. During pattern matching the ? will be overwritten by the pointer to the graph node representing the expression to which the local variable must be bound.

    For simplicity we always reserve place for the maximal number of local variables occurring in a rule of the function corresponding to the environment block.
  - $args \in Adr^*$ is the lists of arguments of the function call. The arguments are represented by pointers to their graph representation.

    We adopt here the handling of arguments in reduction machines. In Warren's machine the arguments are accessed via special argument registers and saved in the choice points if alternative evaluations of a clause are possible. A similar treatment would also be possible in the narrowing machine by replacing the data stack by sets of registers.
  - $sev \in \mathbb{N}$ is the saved pointer to the previous environment block.
  - $ra \in PAdr$ is the return address of the function call, i.e. the program address at which the computation has to be continued after a successful termination of the function call.

  Choice points have the following components

  $$\langle tds, nds, sds, tt, sbp, badr \rangle,$$

  where

  - The components $tds, nds \in \mathbb{N}$ ("top of the data stack, number of saved data stack positions") and $sds \in Adr^*$ ("saved data stack positions") give information of how to restore the data stack on backtracking. The stack has to be deleted up to position tds and then the nds saved entries, sds, have to be copied onto the stack. The management of the choice points and backtracking will be described in detail in the next subsection.
  - $tt \in \mathbb{N}$ indicates the length of the trail to which this must be reset on backtracking. Resetting means unbinding the variables noted in the trail.
  - $sbp \in \mathbb{N}$ is the saved backtrack pointer, i.e. the pointer to the previous choice point.

- $badr \in PAdr$ is called *backtrack address* and indicates the code address of the next alternative rule.

Simple choice points are also required in a reduction machine that does pattern matching explicitely and not by using a pattern matching compiler. These reduction choice points contain the address of the next alternative rule and the original depth of the data stack. They occur only on top of the stack in order to enable the switch to an alternative rule when the matching with a rule fails. As soon as a matching is successful, i.e. an applicable rule has been found, the choice point can be removed from the top of the stack. In the narrowing machine choice points must not be deleted when a rule is applicable, because a successful unification does not imply that no other rule is applicable. Later we will discuss a special situation that allows an early deletion of choice points. The environment stack of the narrowing machine in general contains a mixture of choice points and environments. The top of the stack is always indicated by the maximum of the environment pointer and the backtrack pointer.

- *trail* $tr \in Adr^*$
  The trail is used to mark variable bindings that may have to be reset (undone) if backtracking is necessary.

- *graph* $G : Adr \dashrightarrow GNodes$
  The graph or heap is necessary for the representation of terms, i.e. variables and structured terms. Furthermore the graph may contain special nodes called *black holes* which will be used to construct term representations top down during unification.

  The set *GNodes* of graph nodes contains the following types of nodes:

  - *variable nodes:*
    $\langle \text{VAR}, a\rangle$ with $a \in Adr \cup \{?\}$, where $\langle \text{VAR}, ?\rangle$ represents an unbound variable
  - *constructor nodes:*
    $\langle \text{CONSTR}, c, a_1 : \ldots : a_m\rangle$ with $c \in DC$ and $a_i \in Adr$ $(1 \leq i \leq m)$
  - *black holes:* $\langle \text{HOLE}\rangle$.

The state of the machine will always be given by a tuple of the form

$$(ip, d, st, ep, bp, tr, G) \in Store$$

where

$$Store := PAdr \times Adr^* \times (Adr \cup \mathbb{N} \cup PAdr \cup \{?\})^* \times \mathbb{N}^2 \times Adr^* \times [Adr \dashrightarrow GNodes]$$

and stacks are assumed to grow to the left.

To sum up the narrowing machine has been developed by extending a reduction machine

1. by *variables nodes* in the graph components which are needed to represent the free variables of the goal expressions and their bindings,

2. by the *trail* to keep track of variable binding and finally

3. by the more complex *choice points* on the environment stack.

3.2 Organization of Backtracking

For each function call with several defining rules a choice point is allocated on the environment stack. The choice point contains information to restore the current state of the machine on backtracking: the depth of the data stack, the length of the trail, the previous backtrack pointer and the code address of the next alternative rule which is used to reset the instruction pointer. The environment stack will be saved by the choice point on its top. The environment pointer does not need to be saved as it points at the environment just below the choice point. For simplicity we do not reset the graph on backtracking, although this would be no problem by noting its "depth" in the choice point. As the trail will always grow during forward computations and only shrink on backtracking it is sufficient to store its length in choice points.

Unfortunately, the data stack does not have such a regular behaviour. Due to the nesting of expressions it is possible that the depth of this stack becomes smaller than the depth stored in the last choice point.

Consider e.g. the program rules

$$f(X) := h(a, g(X)) \qquad\qquad h(a,d) := c \qquad\qquad \begin{array}{l} g(a) := c \\ g(b) := d \end{array}$$

and evaluation of the expression $f(X)$. First an environment for the call of f will be generated. Then a pointer to the graph representation of constructor a and a pointer to the unbound variable X will be loaded on the data stack. The call $g(X)$ leads to the generation of an environment for g on the stack. Thereby the pointer to X is deleted from the data stack. As function g allows for two alternative computations a choice point is created. At this time the data stack contains the representation of a and has depth 1. The computation of $g(X)$ using the first rule and binding the variable X to the constructor a is successful and yields a pointer to the graph representation of constructor c on top of the data stack (see figure 1a). Now function h is called with arguments a and c taken from the data stack. Especially the pointer to a that is needed for the alternative computation represented by the choice point is eliminated and must be saved.

In the narrowing machine this will be done by extending the choice point on top of the stack by the part of the data stack that is destroyed but must be saved for backtracking (see figure 1b).

The data stack shrinks when a function call is executed or a new structure is generated in the graph. *Note that the depth of the stack may only become smaller than the depth stored in the last choice point when this is on top of the stack.* Thus it is easy to recognize when a part of the data stack that has to be saved, is destroyed and, as the choice point is on top of the stack, it is easy to save this part of the data stack in the choice point.

When a choice point is created, no data stack entries are saved in the choice point, i.e. $sds = \varepsilon, nds = 0$ and tds is the current depth of the data stack. The tds entries of the data stack must be saved during the further evaluation. Only if the depth of the data stack sinks below the saved depth, data stack entries are saved in the choice point. In this way the size of choice points is kept as small as possible.

In general backtracking will be initiated if a unification fails. The instruction pointer is set to the backtrack address stored in the choice point. The bindings noted in the trail since the generation of the choice point are undone and eliminated. The entries for the local variables within the environment just below the choice point — this environment belongs to the function call that generated the choice point — will be reset to '?'. The environment pointer will be set to point at this environment. To do this resetting the number of local variables is the first entry of each environment. Finally the data stack has to be reset to the depth tds noted in the choice point and the data stack entries saved in the choice point have to be restored on top of the reduced stack. The formal specification of the backtracking operation is given in figure 2.

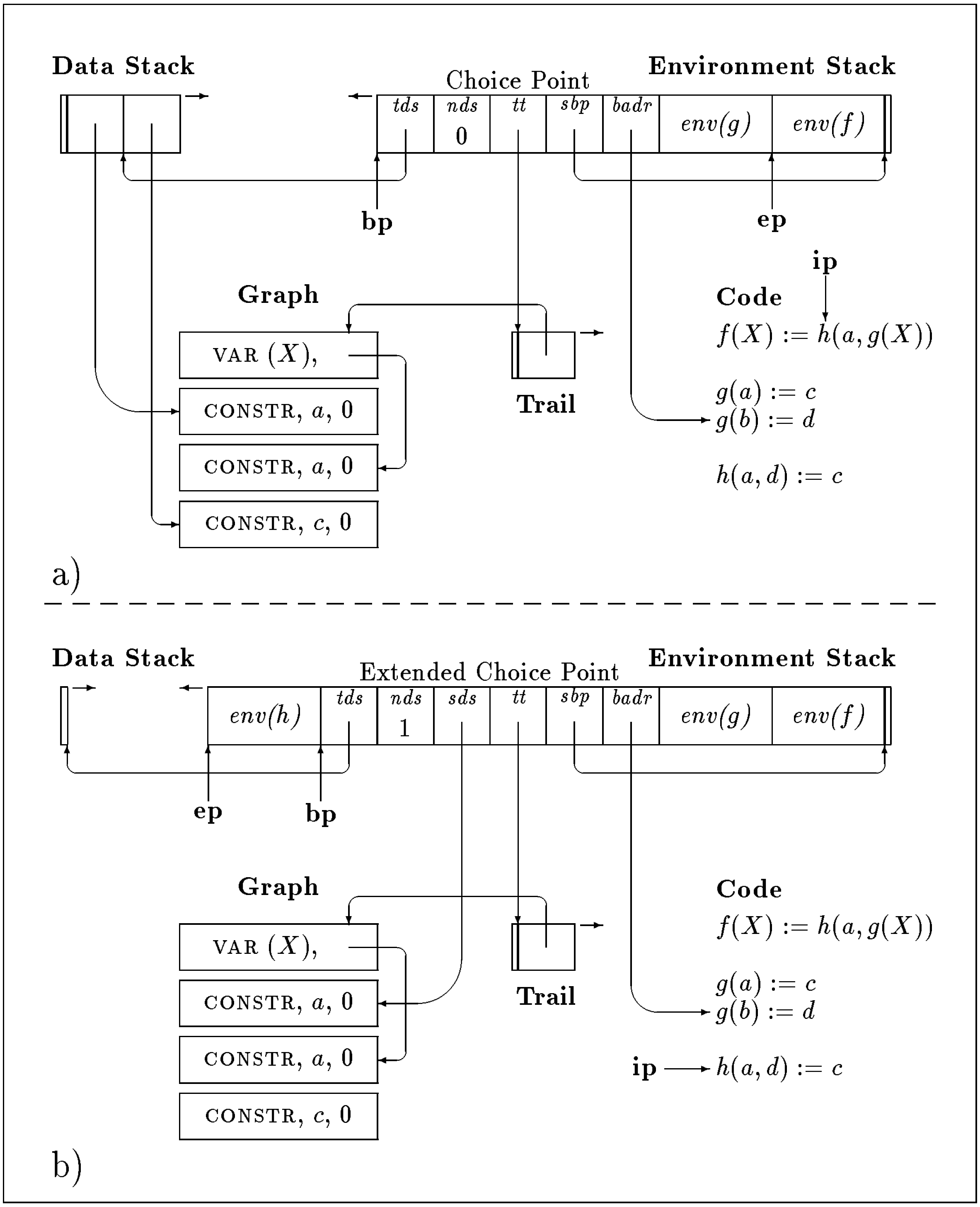

Figure 1: Example — Extension of choice points

$$
\begin{array}{l}
\mathit{backtrack}(ip, d_m : \ldots : d_1, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ st[bp..1] = tds : nds : sds : tt : sbp : badr : k : lv_1 : \ldots : lv_k : a_1 : \ldots : a_m : ep' : ra : st' \\
\qquad \mathit{in}\ (badr, sds : d_{tds} : \ldots : d_1, \\
\qquad\qquad tds : nds : sds : tt : sbp : badr : k :? : \ldots :? : a_1 : \ldots : a_m : ep' : ra : st', \\
\qquad\qquad\qquad bp - nds - 5, bp, tr[1..tt], undo(G, tr[tt..lg(tr)])) \\
\\
\text{where } undo(G, tr) := \mathit{if}\ tr = \varepsilon\ \mathit{then}\ G\ \mathit{else}\ \mathit{let}\ tr = a : tr'\ \mathit{in}\ undo(G[a/\langle \text{VAR}, ?\rangle], tr') \\
\text{and} \quad lg(tr) \text{ denotes the length of the trail } tr.
\end{array}
$$

Figure 2: Backtracking

$$
\begin{array}{l}
\mathcal{C}\ [\![\text{UNIFYVAR } i]\!]\ (ip, d_0 : d, st, ep, bp, tr, G) \\
\quad := \mathit{if}\ G(d_0) = \langle \text{HOLE} \rangle\ \mathit{then}\ (ip+1, d, st[ep-i/d_0], ep, bp, tr, G[d_0/\langle \text{VAR}, ?\rangle]) \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ (ip+1, d, st[ep-i/d_0], ep, bp, tr, G) \\
\\
\mathcal{C}\ [\![\text{UNIFYCONSTR } (c,n)]\!]\ (ip, d_0 : d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ a_1 : \ldots : a_{n+1}\ := new(G, n+1) \\
\qquad \mathit{in}\ \mathit{if}\ G(d_0) = \langle \text{CONSTR}, c, b_1 : \ldots : b_n \rangle\ \mathit{then}\ (ip+1, b_1 : \ldots : b_n : d, st, ep, bp, tr, G) \\
\qquad\quad \mathit{else}\ \mathit{if}\ G(d_0) = \langle \text{VAR}, ? \rangle \\
\qquad\qquad \mathit{then}\ (ip+1, a_2 : \ldots : a_{n+1} : d, st, ep, bp, tr : d_0, \\
\qquad\qquad\quad G[d_0/\langle \text{VAR}, a_1\rangle, a_1/\langle \text{CONSTR}, c, a_2 : \ldots : a_{n+1}\rangle, a_2/\langle \text{HOLE}\rangle, \cdots, a_{n+1}/\langle \text{HOLE}\rangle]) \\
\qquad\quad \mathit{else}\ \mathit{if}\ G(d_0) = \langle \text{HOLE} \rangle \\
\qquad\qquad \mathit{then}\ (ip+1, a_1 : \ldots : a_n : d, st, ep, bp, tr, \\
\qquad\qquad\qquad G[d_0/\langle \text{CONSTR}, c, a_1 : \ldots : a_n\rangle, a_1/\langle \text{HOLE}\rangle, \cdots, a_n/\langle \text{HOLE}\rangle]) \\
\qquad\quad \mathit{else}\ \mathit{backtrack}\ (ip, d_0 : d, st, ep, bp, tr, G) \\
\\
\text{where } new\ (G, n) := \mathit{let}\ a = \min\{adr \in Adr \mid G(adr) \text{ is undefined}\} \\
\qquad\qquad\qquad\qquad \mathit{in}\ \mathit{if}\ n = 1\ \mathit{then}\ a\ \mathit{else}\ a : new(G[a/\langle \text{HOLE}\rangle], n-1)
\end{array}
$$

Figure 3: Unification instructions

3.3 Machine Instructions

The machine instructions are grouped into unification instructions, control instructions and graph instructions.

Unification Instructions

The unification of an argument of a function call and the formal parameter term of a function rule is done by traversing the parameter term top down and performing local comparisons. As a term may be a variable or a constructor term, we distinguish two instructions for local unification steps. The formal specification of these instructions is given in figure 3.

- UNIFYVAR i is used to bind the ith local variable to the argument term represented by the pointer on top of the data stack. It moves the pointer from the data stack to the ith local variable position in the environment. If the pointer refers to a black hole node this is replaced by an unbound variable node.

$$
\begin{array}{l}
\mathcal{C}\ [\![\mathrm{CALL}\ (f,n,k,j)]\!]\ (ip, d_n:\ldots:d_1:d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ top = \max\{ep, bp\}\ \mathit{in} \\
\qquad \mathit{if}\ ep = top\ \mathit{and}\ j > 0 \\
\qquad \mathit{then}\ \%\ \text{tail recursive call} \\
\qquad\qquad \mathit{let}\ st = k' : a'_1 : \ldots : a'_j : ep' : ra' : st'\ \mathit{in} \\
\qquad\qquad (ca(f), d, k : \underbrace{? : \ldots :?}_{k \text{ times}} : d_1 : \ldots : d_n : ep' : ra' : st', ep - j + k + n, bp, tr, G) \\
\qquad \mathit{else}\ \ \mathit{let}\ st = tds : nds : st'\ \mathit{and}\ lg(d) = m \\
\qquad\qquad \mathit{and}\ newenv = k : \underbrace{? : \ldots :?}_{k \text{ times}} : d_1 : \ldots : d_n : ep : ip + 1\ \mathit{in} \\
\qquad\qquad \mathit{if}\ bp = top\ \mathit{and}\ tds > m\ \%\ \text{extension of choice point} \\
\qquad\qquad \mathit{then}\ (ca(f), d, newenv : m : nds + (tds - m) : d_{tds-m} : \ldots : d_1 : st', \\
\qquad\qquad\qquad\qquad\qquad\qquad top + k + n + 3 + (tds - m), bp + (tds - m), tr, G) \\
\qquad\qquad \mathit{else}\ \ (ca(f), d, newenv : st, top + k + n + 3, bp, tr, G),
\end{array}
$$

where $ca(f)$ denotes the code address (the address of the first line of code) of the function f.

$$
\begin{array}{l}
\mathcal{C}\ [\![\mathrm{RET}\ j]\!]\ (ip, d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ st[ep..1] = k : a_1 : \ldots : a_j : ep' : ra : st'\ \mathit{in} \\
\qquad \mathit{if}\ ep > bp\ \mathit{then}\ (ra, d, st', ep', bp, tr, G)\ \mathit{else}\ (ra, d, st, ep', bp, tr, G)
\end{array}
$$

$$
\mathcal{C}\ [\![\mathrm{JMP}\ l]\!]\ (ip, d, st, ep, bp, tr, G) := (l, d, st, ep, bp, tr, G)
$$

$$
\begin{array}{l}
\mathcal{C}\ [\![\left\{\begin{array}{c}\mathrm{JPF}\\ \mathrm{JPT}\end{array}\right\} l]\!]\ (ip, d_0 : d, st, ep, bp, tr, G[d_0/\langle \textsc{constr}, b, \varepsilon\rangle]) \\
\quad := \mathit{if}\ b = \left\{\begin{array}{c}\text{false}\\ \text{true}\end{array}\right\}\ \mathit{then}\ (l, d, st, ep, bp, tr, G)\ \mathit{else}\ (ip+1, d, st, ep, bp, tr, G)
\end{array}
$$

Figure 4: Instructions for the forward control

- UNIFYCONSTR (c, n) is used if the parameter term has the top level constructor c and n component terms. It compares the constructor c with the graph node indicated by the top element of the data stack. If the top element of the data stack points at a constructor node with constructor c and n components, the pointer on top of the stack is replaced by the addresses of the components of this constructor node (pattern matching). The unification with the component terms will be done by the subsequent unification instructions. If the top stack element points at an unbound variable node, this variable is bound to a newly generated c-constructor node. For the n components of this node black holes are constructed. The addresses of these black holes are stored in the constructor node and on top of the stack. If the top element of the stack points at a black hole this node is overwritten by a c-constructor node and again black holes are generated for the components. In all other cases, backtracking is started.

Forward Control Instructions

The forward control instructions are the same as in a reduction machine except that the CALL-instruction may lead to the extension of the top level choice point. The formal specification of these instructions is given in figure 4.

$$
\begin{array}{l}
\mathcal{C}\ [\![\text{TRY\_ME\_ELSE}\ l]\!]\ (ip, d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ top = \max\{bp, ep\}\ \mathit{in}\ (ip+1, d, lg(d) : 0 : lg(tr) : bp : l : st, ep, top+5, tr, G) \\
\\
\mathcal{C}\ [\![\text{RETRY\_ME\_ELSE}\ l]\!]\ (ip, d, tds : nds : sds : tt : sbp : badr : st, ep, bp, tr, G) \\
\quad := (ip+1, d, tds : nds : sds : tt : sbp : l : st, ep, bp, tr, G) \\
\\
\mathcal{C}\ [\![\text{TRUST\_ME\_ELSE\_FAIL}]\!]\ (ip, d, tds : nds : sds : tt : sbp : badr : st, ep, bp, tr, G) \\
\quad := (ip+1, d, st, ep, sbp, tr, G) \\
\\
\mathcal{C}\ [\![\text{BACKTRACK}]\!]\ (ip, d, st, ep, bp, tr, G) := \mathit{backtrack}\ (ip, d, st, ep, bp, tr, G) \\
\\
\mathcal{C}\ [\![\text{POP}]\!]\ (ip, d, st, ep, bp, tr, G) \\
\quad := \mathit{if}\ ep > bp\ \mathit{then}\ (ip+1, d, st, ep, bp, tr, G) \\
\qquad\qquad \mathit{else}\ \mathit{let}\ st = tds : nds : sds : tt : sbp : badr : st' \\
\qquad\qquad\quad \mathit{in}\ \mathit{if}\ lg(tr) = tt\ \mathit{then}\ (ip+1, d, st', ep, bp, tr, G) \\
\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \ (ip+1, d, st, ep, bp, tr, G)
\end{array}
$$

Figure 5: Instructions for the backward control

- The evaluation of new function calls is initiated by the instruction CALL (f, n, k, j). A new environment is put on top of the environment stack taking n pointers to arguments from the data stack and reserving place for k local variables. If the fourth parameter j is different from 0, the instruction overwrites the previous environment if this is on top of the stack (optimized handling of tail recursion). In this case the fourth parameter gives the number of arguments and local variables in the current environment block.

- RET j successfully finishes a function call. The parameter j gives the number of arguments and local variables in the current environment. The instruction pointer is set to the return address and the previous environment pointer is restored. Note that the current environment can only be deleted if it is on top of the stack. If a choice point is on top of the stack the environment will be saved, because it might be needed in an alternative computation.

- JMP l, JPT l, JPF l denote simple and conditional jump instructions.

Backward Control Instructions

The formal specification of the backward control instructions is given in figure 5.

- If a program contains more than one rule for a function symbol, the code for this function starts with the instruction TRY_ME_ELSE l, which has the same meaning as in Warren's machine. A choice point is generated on top of the stack to keep all information necessary to backtrack to the next alternative rule whose code starts at program address l.

- RETRY_ME_ELSE l replaces the backtrack address of the choice point on top of the stack by l if an alternative rule is tried and there are still more alternatives.

- TRUST_ME_ELSE_FAIL is the command that precedes the code generated for the last rule of a function symbol. It eliminates the choice point on top of the stack.

- BACKTRACK immediately leads to backtracking. It is needed for the translation of guarded expressions.

$$\begin{array}{l}
\mathcal{C}\ [\![\text{LOAD } i]\!]\ (ip, d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ a = \mathit{new}(G, 1)\ \mathit{in} \\
\qquad \mathit{if}\ st[ep - i] = \,?\ \mathit{then}\ (ip+1, a : d, st[ep - i/a], ep, bp, tr, G[a/\langle \text{VAR}, ?\rangle]) \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ (ip+1, \mathit{dereference}(G, st[ep-i]) : d, st, ep, bp, tr, G) \\
\\
\text{where } \mathit{dereference}(G, adr) := \mathit{if}\ G(adr) = \langle \text{VAR}, adr' \rangle\ \mathit{then}\ \mathit{dereference}(G, adr')\ \mathit{else}\ adr \\
\\
\mathcal{C}\ [\![\text{NODE } (c,n)]\!]\ (ip, d_n : \ldots : d_1 : d, st, ep, bp, tr, G) \\
\quad := \mathit{let}\ a = \mathit{new}(G, 1)\ \mathit{in} \\
\qquad \mathit{if}\ bp > ep\ \mathit{and}\ st[bp] > lg(d) \\
\qquad \mathit{then}\ \mathit{let}\ st = tds : nds : st'\ \mathit{and}\ lg(d) = m\ \mathit{in} \\
\qquad\qquad (ip+1, a : d, m : nds + (tds - m) : d_{tds-m} : \ldots : d_1 : st', \\
\qquad\qquad\qquad\qquad\qquad\qquad ep, bp + (tds - m), tr, G[a/\langle \text{CONSTR}, c, d_1 : \ldots : d_n\rangle]) \\
\qquad \mathit{else}\ \ (ip+1, a : d, st, ep, bp, tr, G[a/\langle \text{CONSTR}, c, d_1 : \ldots : d_n\rangle])
\end{array}$$

Figure 6: Graph Instructions

- POP eliminates the choice point on top of the stack in special situations. In the reduction machine it is possible to remove the choice point immediately after a successful pattern matching. In the narrowing machine this is of course not possible because several rules might be applicable using different bindings of free variables. If however no free variables have been bound during unification, the nonambiguity of Simple BABEL programs guarantees that no other applicable rule yields a different result. Thus, in the narrowing machine, the instruction POP tests whether new variable bindings have been done, i.e. noted in the trail since the generation of the choice point on top of the stack. If the trail has not grown during the unification, the choice point on top of the stack can be eliminated. Thus, especially locally deterministic computations are recognized and handled as in the functional reduction machine.

Graph Instructions

The formal specification of the graph instructions is given in figure 6.

- LOAD i loads the $(i+1)$th entry (local variable or argument) of the current environment on the data stack. If this entry equals ?, it is replaced by the address of a newly generated unbound variable node and this address is written on the data stack.

- NODE (c, n) generates a new constructor node where n addresses for the components are taken from the data stack and replaced by the address of the newly generated node. If a choice point is on top of the stack and the depth of the data stack becomes smaller than the top element of the environment stack, a part of the stack must additionally be saved in the topmost choice point.

3.4 State Transitions

The transitions of the machine are mainly determined by the code that is generated for a Simple BABEL program. If the goal expression M has k local variables, the machine execution starts with the configuration

$$(ca(M), \varepsilon, k : \underbrace{? : \ldots : ?}_{k \text{ times}} : 0 : 0, k+3, 0, [\,], G_0)$$

a) Let f be a function symbol with arity m, maximally k local variables and more than one defining rules. Then:

$$
\begin{array}{lll}
\multicolumn{3}{l}{proctrans(\langle ft_{i1} \ldots t_{im} = \mathrm{body}_i \mid 1 \leq i \leq r, r \geq 2\rangle)} \\
:= & & \mathrm{TRY\_ME\_ELSE}\ rule_2 \\
 & & ruletrans\ (ft_{11} \ldots t_{1m} = \mathrm{body}_1, k) \\
 & rule_2: & \mathrm{RETRY\_ME\_ELSE}\ rule_3 \\
 & & ruletrans\ (ft_{21} \ldots t_{2m} = \mathrm{body}_2, k) \\
 & rule_3: & \vdots \\
 & & \vdots \\
 & rule_r: & \mathrm{TRUST\_ME\_ELSE\_FAIL} \\
 & & ruletrans\ (ft_{r1} \ldots t_{rm} = \mathrm{body}_r, k)
\end{array}
$$

b) $ruletrans(ft_1 \ldots t_m = \mathrm{body}, k)$

$$
\begin{array}{ll}
:= & \mathrm{LOAD}\ k+1 \\
 & unifytrans\ (t_1) \\
 & \vdots \\
 & \mathrm{LOAD}\ k+m \\
 & unifytrans\ (t_m) \\
 & \mathrm{POP} \\
 & exptrans\ (\mathrm{body}, m+k) \\
 & \mathrm{RET}\ m+k
\end{array}
$$

where $k+i$ is the position of the ith argument in an environment block relative to the environment pointer.

Figure 7: Compilation schemes *proctrans* and *ruletrans*

where $ca(M)$ denotes the address of the first line of code for M and G_0 is assumed to be the empty graph. The transition rule

$$(ip, ds, st, ep, bp, tr, G) \vdash \mathcal{C}[\![ps(ip)]\!](ip, ds, st, ep, bp, tr, G)$$

is then applied until one of the following conditions is true.

- $ep = 0$ (successful computation):
  This indicates that the evaluation has been successful. The result is represented by the top of the data stack while the bindings that have been done are given by the trail and the graph component. If $bp > 0$, more solutions are possible and to obtain these, the machine has to be forced to backtrack.

- $bp = 0$ and $ep > 1$ (failure):
  In this case a failure has occurred and no more choice point is given on the environment stack, i.e. no more alternative computations are possible.

4 Compilation of Simple BABEL Programs

We group the rules of Simple BABEL programs according to the function symbols. Thus a program has the general form:

$$\mathcal{P} = \{\langle f^{(j)} t_{i1}^{(j)} \ldots t_{im}^{(j)} = body_i^{(j)} \mid 1 \leq i \leq r_j\rangle \mid 1 \leq j \leq k\}$$

The code generated for such a program consists of the code for the various procedures (groups of rules for the same function symbol) which will be produced using the scheme *proctrans* given in figure 7a). The defining rules of a function symbol are tested in their textual ordering. Before the first rule is tried, a choice point is put on top of the environment stack to keep note of the alternative rules. This choice point always contains the code address of the next rule. It can be removed, if the last rule is applied.
If there exists only a single rule for the symbol f, the code for the procedure corresponds to the code produced for this rule by the *ruletrans* scheme.

The translation of each rule consists of code for the unification of the arguments of the function application with the terms on the left hand side of the rule and code for the evaluation of the body. After the unification phase the POP instruction tests whether the computation is deterministic (i.e.

$$unifytrans\ (X_i) := \text{UNIFYVAR}\ i$$

$$\begin{array}{ll} \multicolumn{2}{l}{unifytrans\ (c(t_1, \ldots, t_n))} \\ := & \text{UNIFYCONSTR}\ (c, n) \\ & unifytrans\ (t_1) \\ & \vdots \\ & unifytrans\ (t_n) \end{array}$$

$$exptrans\ (X_i, j) := \text{LOAD}\ i$$

$$\begin{array}{ll} \multicolumn{2}{l}{exptrans\ (c(M_1, \ldots, M_n), j)} \\ := & exptrans\ (M_1, 0) \\ & \vdots \\ & exptrans\ (M_n, 0) \\ & \text{NODE}\ (c, n) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{exptrans\ (f(M_1, \ldots, M_n), j)} \\ := & exptrans\ (M_1, 0) \\ & \vdots \\ & exptrans\ (M_n, 0) \\ & \text{CALL}\ (f, n, k, j) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{exptrans\ (B \to M, j)} \\ := & exptrans\ (B, 0) \\ & \text{JMT lb\_M} \\ & \text{BACKTRACK} \\ \text{lb\_M:} & exptrans\ (M, j) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{exptrans\ (B \to M \square N, j)} \\ := & exptrans\ (B, 0) \\ & \text{JMF lb\_N} \\ & exptrans\ (M, j) \\ & \text{JMP end\_lb} \\ \text{lb\_N:} & exptrans\ (N, j) \\ \text{end\_lb:} & \ldots \end{array}$$

Figure 8: Compilation schemes *unifytrans* and *exptrans*

a choice point is on top of the stack and since the generation of this choice point no variable bindings have been done) and in that case eliminates the choice point on top of the stack. In the code for the last rule of a function and if there is only one rule, the POP instruction is superfluous, because the choice point has already been deleted by the TRUST_ME_ELSE_FAIL instruction or no choice point has been created. It should therefore be omitted in these cases.

The translation schemes given in figure 8 are used to produce code for the unification and the evaluation of expressions:

- *unifytrans* : *Term* $\to$ *Code* generates code, which unifies an argument of the actual task given on top of the data stack with the corresponding term on the right hand side of a rule.

- *exptrans* : *Exp* $\times\ \mathbb{N} \to$ *Code* produces code, which evaluates an expression to normal form (in particular the right hand side of a rule and the goal expression). The second argument indicates whether a tail recursive function call is possible. If this argument is 0, we do not have tail recursion. This is the case for the goal expression. If it is different from 0, it gives the number of argument and local variable positions in the current environment that can be overwritten by the environment of the tail recursive call.

  Note that the translation scheme given in figure 8 realizes the innermost evaluation strategy.

Of course, one should optimize code sequences by avoiding sequences of the form LOAD i; UNIFY-VAR j by directly writing LOAD i in the code for the body of the function call. This also decreases the number of local variable locations in the environment. After this simple optimization we get the code sequence given in figure 9 for the small example program of subsection 3.2.

'f':	NODE $(a,0)$	'g':	TRY_ME_ELSE $rule_2$
	LOAD 1		LOAD 1
	CALL $(g,1,0,0)$		UNIFYCONSTR $(a,0)$
	CALL $(h,2,0,1)$		POP
	RET 1		NODE $(c,0)$
'h':	LOAD 1		RET 1
	UNIFYCONSTR $(a,0)$	'$rule_2$':	TRUST_ME_ELSE_FAIL
	LOAD 2		LOAD 1
	UNIFYCONSTR $(d,0)$		UNIFYCONSTR $(b,0)$
	NODE $(c,0)$		NODE $(d,0)$
	RET 2		RET 1

Figure 9: Code for the example program of subsection 3.2

5 Extensions

BABEL is a higher order polymorphically typed functional logic language which uses lazy narrowing as evaluation mechanism. For short, the relationship of Simple BABEL and BABEL can be described by the following equation:

$$\begin{array}{rll} \text{BABEL} = & \text{Simple BABEL} & \\ & + & \text{polymorphic types} \\ & + & \text{higher-order functions} \\ & + & \text{lazy narrowing} \\ & + & \text{free variables in guards on rhs of rules} \end{array}$$

The last extension concerns the form of the function rules. BABEL's function rules have a slightly more general form than the rules of Simple BABEL. A BABEL rule can be guarded by a boolean expression which may contain free variables, i.e. variables not occuring in the left hand side of the rule:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{lhs}} := \underbrace{\underbrace{\{B \rightarrow\}}_{\text{optional guard}} \quad \underbrace{M}_{\text{body}}}_{\text{rhs}}$$

Note that occurrences of free variables are allowed in the guard of a rule, but not in the body. This restriction is necessary to guarantee determinacy. Furthermore the *nonambiguity condition* is generalized in the following way:

> Given any two rules for the same function symbol f:
>
> $$f\ t_1 \ldots t_n := \{B \rightarrow\} M \quad \text{and} \quad f\ s_1 \ldots s_n := \{C \rightarrow\} N$$
>
> one of the three following cases must hold:
>
> (a) *No superposition*: $f\ t_1 \ldots t_n$ and $f\ s_1 \ldots s_n$ are not unifiable.
>
> (b) *Fusion of bodies*: $f\ t_1 \ldots t_n$ and $f\ s_1 \ldots s_n$ have a most general unifier (m.g.u.) σ such that $M\sigma, N\sigma$ are identical.
>
> (c) *Incompatibility of guards*: $f\ t_1 \ldots t_n$ and $f\ s_1 \ldots s_n$ have a m.g.u. σ such that the conjunction $(B \wedge C)\sigma$ is incoherent.
>
> *Incoherence* is defined as a decidable syntactical property of expressions, and chosen in such a way that no incoherent expression may denote the boolean value true, cfr. [Moreno, Rodríguez 89].

The guarded rules allow a simple translation of Prolog clauses into BABEL. The body of the clause becomes the guard of the BABEL rule whose body is identical to true. To ensure left linearity the guard must be extended by appropriate equality conditions.

Considering the above equation the following three extensions are necessary to implement full BABEL on the stack narrowing machine. Note that polymorphic types have only to be handled by the compiler. In the machine we assume that the compiled programs are correctly typed.

5.1 Higher Order Functions

BABEL supports higher order functions in the same way as they are used in functional languages. Higher order *logic* variables are *not* allowed: higher order variables may occur in the lhs of rules, but are forbidden to occur free in either rhs of rules or goals. This means that they are used only for rewriting, as in applicative functional programming.

Thus, higher order functions can simply be supported by introducing function nodes in the graph which will contain the name or code address of a function $f \in FS^k$, a partial list of arguments $a_1 : \ldots : a_m$ with $m < k$, the number of missing arguments $m - k$ and the maximal number of local variables in the program rules for f:

$$\langle \text{FUNCTION}, f, a_1 : \ldots : a_m, k - m, nlv \rangle.$$

Function nodes will be created by a new instruction FNODE, which takes the function symbol and the number of given arguments as parameters. A new instruction APPLY performs the application of a function represented by a function node to further arguments. It behaves as the CALL instruction, if the further arguments complete the application, and as the FNODE instruction if the number of arguments does not yet correspond to the arity of the function symbol. The scheme *exptrans* must be extended by the following cases:

$$\begin{aligned} \mathit{exptrans}\ ((MN), j) := \ & \mathit{exptrans}\ (M, j) \\ & \mathit{exptrans}\ (N, j) \\ & \text{APPLY} \end{aligned} \qquad\qquad \mathit{exptrans}\ (f, j) := \text{FNODE}\ (f, 0)$$

5.2 Lazy Narrowing

BABEL works with a *lazy* narrowing strategy. Hence, it tries to narrow expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the pattern in the lhs of some rule) and contributes to some later narrowing step at an outer position.

Initiating argument evaluations during unification contains a risk of nontermination, assuming that backtracking is done in the reverse order of forward computation. Lets call this strategy *straightforward lazy narrowing*. Consider the following example program which is due to Juanjo Moreno Navarro:

$$\begin{aligned} one(0) &:= s(0) \\ one(s(X)) &:= one(X) \\ \\ nth(0, [Y|Ys]) &:= Y \\ nth(s(X), [Y|Ys]) &:= nth(X, Ys) \\ \\ nats(X) &:= [X|nats(s(X))] \end{aligned}$$

and the goal expression

$$nth(one(X), nats(0)).$$

Note that the function *one* will always yield the value $s(0)$ for all possible argument values $s^i(0)$. Thus the semantics of the goal expression is the second element of the list $nats = [0, s(0), \ldots]$, i.e. the term $s(0)$. Innermost narrowing will yield this value.

Straightforward lazy narrowing of the goal expression will however not terminate. Trying the first rule of *nth* will force the evaluation of the first argument $one(X)$ which will yield the value $s(0)$ binding the free variable X to 0. As it is not possible to unify 0 and $s(0)$, backtracking is initiated and $one(X)$ is again evaluated yielding the value $s(0)$ while binding X to the term $s(0)$. Again the first rule of *nth* is not applicable and backtracking is initiated. This process will not terminate, because there are infinitely many alternative evaluations of $one(X)$ but none will allow to apply the first rule of *nth*.

To avoid such cases of nontermination it is better to try all rules of a function (finitely many!) with an evaluated argument before backtracking on the argument evaluation. Instead of changing the backtracking mechanism it is however possible to do a program transformation of functional logic programs into so called *uniform programs*.

A BABEL program is called *uniform*, if for each $f \in FS^n$ with program rules

$$\begin{array}{llll} f\ t_{11} \ldots & \boxed{t_{1j}} & \ldots t_{1n} := rhs_1 \\ \vdots & \vdots & \vdots \\ f\ t_{r1} \ldots & \boxed{t_{rj}} & \ldots t_{rn} := rhs_r \end{array}$$

the following conditions hold:

1. $\forall j\ (\forall i\ t_{ij} \in Var \vee \underbrace{\forall i\ t_{ij} \notin Var}_{\Rightarrow\ j\text{th argument is demanded}})$

2. $\forall j\ \forall i\ t_{ij}$ is a flat term, i.e. it is a variable or a constructor term whose components are variables.

In [Moreno et al. 90] an automatic transformation of BABEL programs into *uniform programs* is specified. This transformation does not introduce significant inefficiencies and has a number of advantages: Demanded arguments can be easily detected and evaluated to head normal form (*hnf*) before trying to apply rules by unification. This ensures that all rules are tried for a fixed *hnf* evaluation of demanded arguments before backtracking for arguments' reevaluation is activated, which tends to avoid the nontermination problems of straightforward lazy narrowing.

The implementation of uniform BABEL programs on our stack machine causes no problems. For demanded arguments the innermost evaluation strategy can be taken. Non-demanded arguments are represented by newly introduced *suspension nodes*, which contain the code address of the argument and the environment that is needed during the execution of this code, i.e. a copy of the lists of local variables and arguments within the environment block that is active when the suspension node is created. Note that these lists contain only graph pointers. Furthermore, the node contains place to keep note of the result after a successful evaluation. The node must not be overwritten after its evaluation, because backtracking may lead to a reset.

$$\langle \text{SUSPENSION}, \textit{ca}, \textit{locvars}, \textit{arguments}, \textit{result pointer} \rangle.$$

Evaluation of a suspension node leads to the creation of a new environment block on top of the stack, execution of the code at address *ca* and finally an update that notes the result of the evaluation in the suspension node and adds the address of the suspension node to the trail. A formal description of this mechanism will be given in a forthcoming paper.

5.3 Free Variables in Guards

The translation of the rhs of guarded rules can be done similar to the translation of guarded expressions. The POP instruction must however be placed just before the evaluation of the body, because a rule is only applicable if the unification is successful and the guard can be evaluated to true. The generalized nonambiguity condition guarantees that no alternative rule needs to be tried if the current rule is applicable and no variable binding has been noted in the trail during unification and evaluation of the guard. The POP instruction may then delete all the environments and choice points on top of the current environment. This situation is however not realistic, because in general the free variables of the guard will be bound and trailed during its evaluation. By a special treatment of free guard variables it is possible to detect further situations that allow the elimination of choice points. E.g. it is possible to delete the choice points and environments of a guard evaluation, if only the free variables of the guard have been bound during its evaluation.

6 State of the Implementation

The specification of the narrowing machine has been formulated in Miranda to get a first impression of the behaviour of the machine. This prototype implementation shows that the elimination of choice points by the instruction POP has the effect that purely functional computations are executed almost in the same way as in reduction machines. Backtracking is performed very efficiently. As stacks are realized as Miranda lists, the time behaviour of the implementation does not allow to draw final conclusions. We currently develop a more efficient implementation in C where some extensions will be included. For the evaluation of arithmetic expressions we will provide a direct representation of numbers and an implicit implementation of the arithmetic operations. Up to now the equality operator has been implemented by implicit rules which is not very efficient. The final implementation of the machine will contain an explicit equality check (similar to the one specified in [Kuchen et al. 90]).

7 Related Work

Another approach to the implementation of functional logic languages based on Warren's Prolog Engine has been presented in [Balboni et al. 89], [Bosco et al. 89]. In these papers, programs are transformed into a flat form that allows the use of SLD resolution as evaluation mechanism. Thus, narrowing is reduced to SLD resolution and, at least for an innermost evaluation strategy, Warren's machine can be used without any extension. For a lazy implementation an extension of Warren's machine has been proposed [Bosco et al. 89].

The transformation into flat form introduces a new variable for each nested expression. The direct handling of nested expressions in our approach is more space efficient. Additional space is only necessary for the extension of choice points, when it is unavoidable.

The lazy narrowing machine of [Bosco et al. 89] implements the straightforward lazy narrowing strategy that may lead to nontermination in cases where the innermost strategy terminates.

Our stack narrowing machine leads to a simple dynamic detection of determinate computations and thus supports purely functional computations in a good way. Transforming functional logic programs into logic programs and using SLD-resolution for their evaluation hides the nature of the functional components of programs and therefore may make an optimized handling of functional computations more difficult.

[Lindstrom 87] describes the extension of a distributed graph reduction machine for functional languages by features that support logical variables while preserving lazy evaluation, concurrency opportunities and global determinacy. However, Or-parallelism and backtracking are not supported.

The extension of a graph reduction machine by features that support unification and backtracking has been developed in [Kuchen et al. 90]. The backtracking mechanism of this graph machine is more complicated due to the decentralized organization of the control information in the graph structure. The advantage of taking a graph structure instead of a stack lies in the opportunity to exploit parallelism in a more appropriate way, as has been shown in [Loogen et al. 89] for purely functional languages. A lazy graph narrowing machine has been developed in [Moreno et al. 90].

8 Conclusions and Future Work

Our narrowing machine is an amalgamation of a stack reduction machine for functional languages and Warren's Prolog Engine where we omitted several optimizations to obtain a simple presentation of the new implementation technique. To support nested expressions directly, i.e. without program transformations, one needs to extend choice points dynamically. As such an extension is only compelling when the choice point that has to be extended is on top of the stack, no technical problems arise. By taking into account the nonambiguity of functional logic programs it is possible to recognize locally deterministic computations by a simple runtime check and thus to treat them in an optimized way.

In a forthcoming paper, we will give a complete description of the implementation of full BABEL, i.e. of a higher order lazy functional language using our stack narrowing machine. In addition to the development of a more appropriate implementation, it has to be investigated to what extent the stack narrowing machine can be embedded in a distributed system in order to exploit parallelism.

Acknowledgements

The author thanks Herbert Kuchen, Juanjo Moreno Navarro and Mario Rodríguez Artalejo for lots of interesting discussions on narrowing and its implementation. She is also very grateful to David de Frutos Escrig who suggested to take advantage of the nonambiguity of programs in an implementation. Stephan Winkler developed a complete BABEL programming system in Miranda using the stack narrowing machine described in this paper.

References

[Balboni et al. 89] G.P.Balboni, P.G.Bosco, C.Cecchi, R.Melen, C.Moiso, G.Sofi: *Implementation of a Parallel Logic Plus Functional Language*, in: P.Treleaven (ed.), Parallel Computers: Object Oriented, Functional and Logic, Wiley 1989.

[Bellia, Levi 86] M. Bellia, G. Levi: *The Relation between Logic and Functional Languages*, Journal of Logic Programming, Vol.3, 1986, 217–236.

[Bosco et al. 89] P.G.Bosco, C.Cecchi, C.Moiso: *An extension of WAM for K-LEAF: A WAM-based compilation of conditional narrowing*, Int. Conf. on Logic Programming, Lisboa, 1989.

[DeGroot, Lindstrom 86] D.DeGroot, G.Lindstrom (eds.): *Logic Programming: Functions, Relations, Equations*, Prentice Hall 1986.

[Field, Harrison 88] A.J.Field, P.G.Harrison: *Functional Programming*, Addison-Wesley 1988.

[Kuchen et al. 90] H.Kuchen, R.Loogen, J.J. Moreno-Navarro, M.Rodríguez-Artalejo: *Graph-based Implementation of a Functional Logic Language*, European Symposium on Programming 1990, LNCS 432, Springer Verlag 1990.

[Lindstrom 87] G.Lindstrom: *Implementing logical variables on a graph reduction architecture*, Workshop on Graph Reduction, LNCS 279, Springer Verlag 1987, 382-400.

[Loogen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Distributed Implementation of Programmed Graph Reduction*, Conf. on Parallel Architectures and Languages Europe 1989, LNCS 365, Springer Verlag 1989.

[Moreno, Rodríguez 88] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *BABEL: A functional and logic programming language based on constructor discipline and narrowing*, Conference on Algebraic and Logic Programming 1988, LNCS 343, Springer Verlag 1989.

[Moreno, Rodríguez 89] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Technical Report DIA/89/3, Universidad Complutense, Madrid 1989, to appear in the Journal of Logic Programming.

[Moreno et al. 90] J.J.Moreno-Navarro, H.Kuchen, R.Loogen, M.Rodríguez-Artalejo: *Lazy Narrowing in a Graph Machine*, Conference on Algebraic and Logic Programming 1990, LNCS 463, Springer Verlag 1990.

[Reddy 85] U.S.Reddy: *Narrowing as the Operational Semantics of Functional Languages*, IEEE Int. Symp. on Logic Programming, IEEE Computer Society Press, July 1985, 138–151.

[Reddy 87] U.S.Reddy: *Functional Logic Languages, Part I*, Workshop on Graph Reduction, LNCS 279, Springer Verlag 1987, 401–425.

[Warren 83] D.H.D.Warren: *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, Menlo Park, California, October 1983.