Stack-based Implementation of Narrowing

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

Aachener Informatik–Berichte
Nr. 90–4

Abstract

An abstract stack machine for the implementation of narrowing is presented. On the one hand, this machine can be seen as an extension of a stack-based reduction machine for purely functional languages by components that are necessary for the realization of unification and backtracking. On the other hand, the machine represents an extension of Warren's Prolog engine [Warren 83] that enables the 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 developped in Miranda.

Contents

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 can be viewed as an extension of a stack-based reduction machine by components that are necessary for the realization of unification and backtracking. On the other hand, the machine is an extension of Warren's Prolog engine by features that enable the handling of nested expressions. 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.

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 narrowing machine. The compilation of Simple BABEL-programs into machine code is specified in section 4. In section 5 we report on the current state of the implementation and possible extensions. A discussion of related work is given in section 6. Section 7 finally contains some conclusions.

---

*Miranda is a trademark of Research Software Limited.

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 untyped functional logic language based on a constructor discipline. Its operational semantics is innermost narrowing. The restriction to Simple BABEL has been done to simplify the explanation and specification of the narrowing machine. Extensions of the narrowing machine 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* $s, t, \ldots \in Term$:
$$\begin{array}{rcll} t & ::= & X & \%\ \text{Variable} \\ & | & c(t_1, \ldots, t_n) & \%\ c \in DC^n, n \geq 0 \end{array}$$
- *Expressions* $M, N \ldots \in Exp$:
$$\begin{array}{rcll} M & ::= & t & \%\ t \in Term \\ & | & c(M_1, \ldots, M_n) & \%\ c \in DC^n, n \geq 0 \\ & | & f(M_1, \ldots, M_n) & \%\ f \in FS^n, n \geq 0 \\ & | & (B \to M) & \%\ \text{guarded expression} \\ & | & (B \to M_1 \Box M_2) & \%\ \text{conditional expression} \end{array}$$

$B \to M$ 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 functions 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. *Flatness*: $t_i \in Term$.
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:

¬ false	:=	true
¬ true	:=	false

false ∧ Y	:=	false
true ∧ Y	:=	Y

false ∨ Y	:=	Y
true ∨ 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	% $c \in DC^n$,$d \in DC^m$,
		% $c \neq d$ oder $n \neq m$

2.2 Narrowing

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

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

$$\longrightarrow_\sigma \subseteq \mathit{Exp} \times \mathit{Exp} \text{ mit } \sigma : \mathit{Var} \to \mathit{Exp}$$

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

1. $(\text{true} \to M) \longrightarrow_\varepsilon M$ $\qquad$ $(\text{true} \to M_1 \Box M_2) \longrightarrow_\varepsilon M_1$
   $(\text{false} \to M_1 \Box M_2) \longrightarrow_\varepsilon M_2$

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 $\theta \cup \sigma : \mathit{Var} \to \mathit{Exp}$ 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 which corresponds to the determination of a substitution $\theta : \mathit{Var} \to \mathit{Exp}$ with $t_i\theta = M_i$, is replaced by *unification* of 'pattern' t_i with expressions M_i, where we explicitly allow 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
  - $(B \rightarrow M) \Longrightarrow_\sigma (B' \rightarrow M\sigma)$
  - $(B \rightarrow M_1 \Box M_2) \Longrightarrow_\sigma (B' \rightarrow M_1\sigma \Box M_2\sigma)$

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 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 only the *leftmost innermost* narrowing strategy.

3 The Narrowing Machine

3.1 Components of the Store

The store of the narrowing machine contains the following components:

- *program store* $ps : PAdr - \rightarrow Instr$
  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 data manipulations. The stack entries are graph (heap) addresses, $Adr := \mathbb{N}$.

- *(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. Furthermore, control information (choice points) is stored in order to keep track of possible alternative computations. Environments

contain the control information for forward computations while choice points control backward computations, i.e. backtracking.

The access to the environment stack is achieved 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 are structured in the following way

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

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. At the beginning of a function call these positions are initialized with the symbol ? to indicate that a binding has not yet occurred. Thus, ? is a shorthand for a pointer to an unbound variable node. On binding, the ? will be overwritten by the pointer to the node representing the expression to which the local variable must be bound.

  For simplicity we do not consider 'trimming of environments' as it is done in 'Warren's Prolog engine' but 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 is also 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

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

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 length of the trail to which this must be reset on backtracking. Resetting means unbinding the variables noted in the trail.
  - $lbp \in \mathbb{N}$ is the previous backtrack pointer, i.e. the pointer to the previous choice point.
  - $badr \in PAdr$ is called *backtrack address* and indicates the address of the alternative code.

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

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

  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

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

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 for the next alternative rule in order 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 the stack becomes smaller than the depth stored in the last choice point.
Consider e.g. the program rules

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

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. Then 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 stack that is destroyed but must be saved for backtracking (see figure 1b).

The stack only 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 sink beyond 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 stack entries are saved in the choice point, i.e. $sds = \varepsilon, nds = 0$ and tds is the current depth of the stack. The tds entries of the 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 an 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 resettingthe 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 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.

3.3 Machine Instructions

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

Unification Instructions

- UNIFYVAR i copies the pointer on top of the stack to the ith local variable in the environment. If this pointer represents a black hole this is replaced by an unbound variable node.

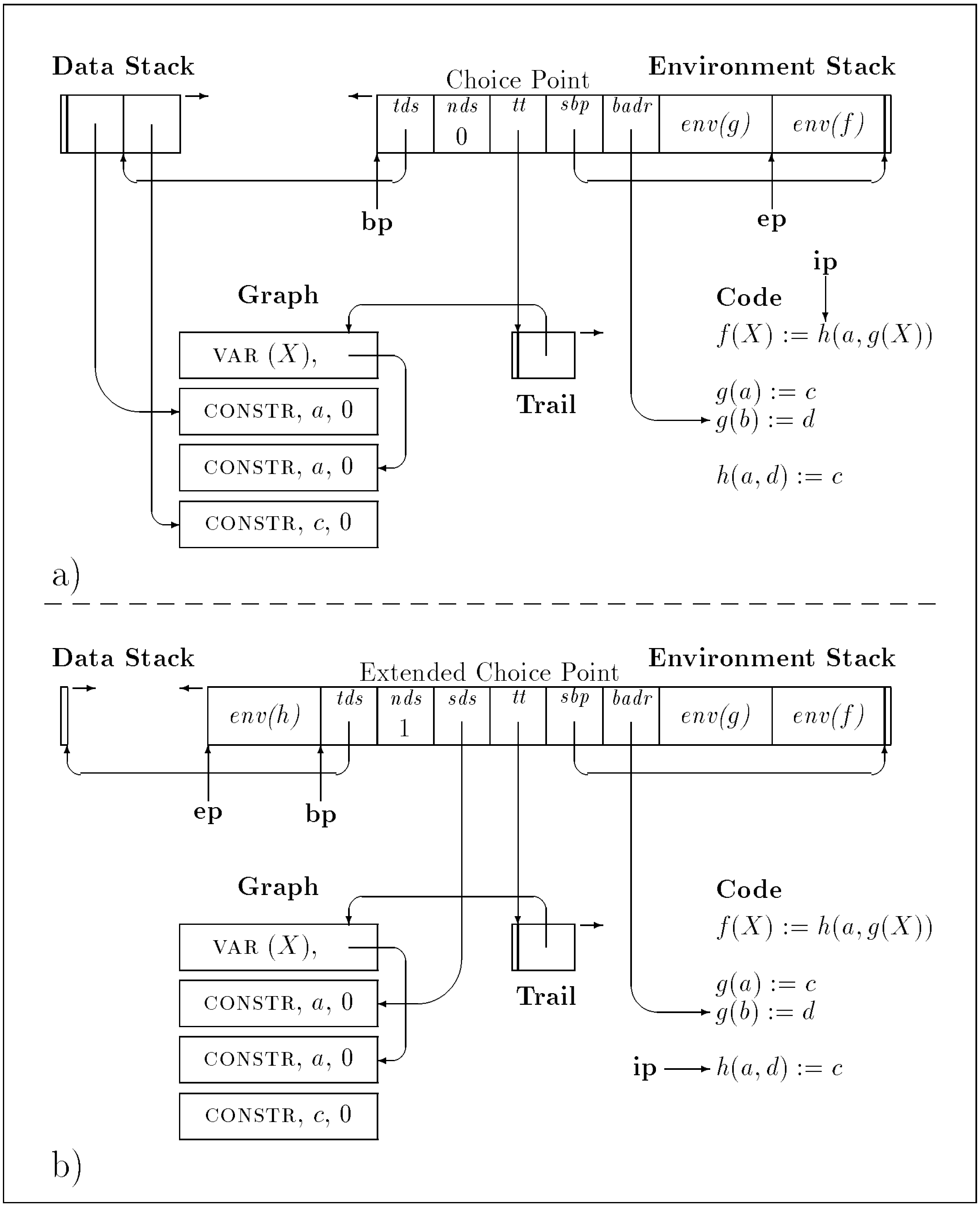

Figure 1: Example — Extension of choice points

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

Figure 2: Backtracking

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

Figure 3: Unification instructions

- UNIFYCONSTR (c, n) compares the constructor c with the graph node represented by the top element of the data stack. If the top element of the data stack points at a constructor node with constructor c, the pointer on top of the stack is replaced by the components of this constructor node. If it points at an unbound variable node, this variable is bound to a newly generated c-constructor node. For the 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.

The formal specification of the unification instructions is given in figure 3.

Forward Control Instructions

- 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

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

where $ca(f)$ denotes the code address of f, i.e. the address of the first line of code for 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

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

The formal specification of the forward control instructions is given in figure 4.

Backward Control Instructions

- TRY_ME_ELSE l 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 whose code starts at program address l.

$$\mathcal{C}\ [\![\text{TRY\_ME\_ELSE}\ l]\!]\ (ip, d, st, ep, bp, tr, G)$$
$$:= let\ top = \max\{bp, ep\}$$
$$in\ (ip+1, d, lg(d) : 0 : lg(tr) : bp : l : st, ep, top+5, tr, G)$$

$$\mathcal{C}\ [\![\text{RETRY\_ME\_ELSE}\ l]\!]\ (ip, d, tds : nds : sds : tt : lbp : badr : st, ep, bp, tr, G)$$
$$:= (ip+1, d, tds : nds : sds : tt : lbp : l : st, ep, bp, tr, G)$$

$$\mathcal{C}\ [\![\text{TRUST\_ME\_ELSE\_FAIL}]\!]\ (ip, d, tds : nds : sds : tt : lbp : badr : st, ep, bp, tr, G)$$
$$:= (ip+1, d, st, ep, lbp, tr, G)$$

$$\mathcal{C}\ [\![\text{FORCE}]\!]\ (ip, d, st, ep, bp, tr, G) := backtrack\ (ip, d, st, ep, bp, tr, G)$$

$$\mathcal{C}\ [\![\text{TEST}]\!]\ (ip, d, tds : nds : sds : tt : lbp : badr : st, ep, bp, tr, G)$$
$$:= if\ lg(tr) = tt\ then\ (ip+1, d, st, ep, bp, tr, G)$$
$$else\ \ (ip+1, d, tds : nds : sds : tt : lbp : l : st, ep, bp, tr, G)$$

Figure 5: Instructions for the backward control

- RETRY_ME_ELSE l replaces the backtrack address of the choice point on top of the stack by l.
- TRUST_ME_ELSE_FAIL is the first command of the code generated for the last rule of a function symbol. It eliminates the choice point on top of the stack.
- FORCE immediately leads to backtracking.
- TEST 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. It will be executed after a successful unification phase during the application of a rule. If the trail has not grown during the unification, this rule is the only applicable rule due to the nonambiguity property of Simple BABEL programs and the choice point on top of the stack can be eliminated.

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

Graph Instructions

- 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 replacing n addresses on the data stack 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 then the top element of the environment stack, a part of the stack must additionally saved in the topmost choice point.

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

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

Figure 6: Graph Instructions

3.4 State Transitions

The transitions of the machine are mainly determined by the code that is generated for a Simple BABEL program. For simplicity we consider here only goals of the form $f(t_1, \ldots, t_m)$ where f is a function of the program. If the rules for f have maximally k local variables the machine execution starts with the configuration

$$(ca(f), \varepsilon, k : \underbrace{? : \ldots : ?}_{k \text{ times}} : a_1 : \ldots : a_m : 1 : 0, m + k + 4, 0, G_0)$$

where $ca(f)$ denotes the address of the first line of code for f and $G_0(a_i)$ is assumed to be the root node of the graph representation of t_i in G_0 $(1 \leq i \leq m)$. 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 are true.

- $ep = 1$ (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 computation is 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*.

For a function symbol f with program-arity m, maximally k local variables and more than one defining rules

$$ft_{i1} \dots t_{im} = \text{body}_i \quad (1 \le i \le r, r \ge 2)$$

the following code will be generated by the scheme *proctrans*:

$$\begin{array}{ll}
 & \text{TRY_ME_ELSE } \mathit{rule}_2 \\
 & \mathit{ruletrans}\ (ft_{11} \dots t_{1m} = \text{body}_1, k) \\
\mathit{rule}_2: & \text{RETRY_ME_ELSE } \mathit{rule}_3 \\
 & \mathit{ruletrans}\ (ft_{21} \dots t_{2m} = \text{body}_2, k) \\
\mathit{rule}_3: & \qquad\vdots \\
 & \qquad\vdots \\
\mathit{rule}_r: & \text{TRUST_ME_ELSE_FAIL} \\
 & \mathit{ruletrans}\ (ft_{r1} \dots t_{rm} = \text{body}_r, k)
\end{array}$$

The defining rules of a function symbol are tested in their textual ordering. 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 code for the unification phase the TEST-instruction tests whether the computation is deterministic (i.e. 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.

$$\mathit{ruletrans}(ft_1 \dots t_m = \text{body}, k)$$

produces the following code:

$$\begin{array}{l}
\text{LOAD } k+1 \\
\mathit{unifytrans}\ (t_1) \\
\qquad\vdots \\
\text{LOAD } k+m \\
\mathit{unifytrans}\ (t_m) \\
\text{TEST} \\
\mathit{exptrans}\ (\text{body}, m+k) \\
\text{RET } m+k
\end{array}$$

where $k+i$ is the position fo the ith argument in an environment block relative to the environment pointer.
The following translation schemes are used to produce code for the unification and the evaluation of expressions:

- *unifytrans* : $\mathbf{Term}_\Sigma \to \mathit{Code}$ 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.

$$\begin{array}{l} \mathit{unifytrans}\ (X_i) := \text{UNIFYVAR}\ i \\ \mathit{unifytrans}\ (c(t_1, \ldots, t_n)) := \text{UNIFYCONSTR}\ (c, n) \\ \qquad\qquad\qquad\qquad\qquad \mathit{unifytrans}\ (t_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad \mathit{unifytrans}\ (t_n) \end{array}$$

- *exptrans* : $\mathbf{Exp}_\Sigma \times \mathbb{N} \rightarrow \mathit{Code}$ produces code, which evaluates an expression to normal form (in particular the right hand side of a rule). The second argument indicates whether a tail-recursive function call is possible. If this argument is 0, we do not have tail recursion. 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 following translation realizes the innermost evaluation strategy.

$$\begin{array}{l} \mathit{exptrans}\ (X_i, j) := \text{LOAD}\ i \\ \\ \mathit{exptrans}\ (c(M_1, \ldots, M_n), j) \\ \quad := \ \mathit{exptrans}\ (M_1, 0) \\ \qquad\qquad \vdots \\ \qquad\quad \mathit{exptrans}\ (M_n, 0) \\ \qquad\quad \text{NODE}\ (c, n) \\ \\ \mathit{exptrans}\ (f(M_1, \ldots, M_n), j) \\ \quad := \ \mathit{exptrans}\ (M_1, 0) \\ \qquad\qquad \vdots \\ \qquad\quad \mathit{exptrans}\ (M_n, 0) \\ \qquad\quad \text{CALL}\ (f, n, k, j) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{exptrans}\ (B \rightarrow M, j)} \\ := & \mathit{exptrans}\ (B, 0) \\ & \text{JMT lb\_M} \\ & \text{FORCE} \\ \text{lb\_M:} & \mathit{exptrans}\ (M, j) \\ \\ \multicolumn{2}{l}{\mathit{exptrans}\ (B \rightarrow M \Box N, j)} \\ := & \mathit{exptrans}\ (B, 0) \\ & \text{JMF lb\_N} \\ & \mathit{exptrans}\ (M, j) \\ & \text{JMP end\_lb} \\ \text{lb\_N:} & \mathit{exptrans}\ (N, j) \\ \text{end\_lb:} & \ldots \end{array}$$

Of course, one should optimize code sequences by avoiding sequences of the form LOAD i; UNIFYVAR 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. Furthermore the instruction TEST can be omitted when we have only one rule for a function symbol. After these simple optimizations we get the code sequence given in figure 7 for the small example program of subsection 3.2.

5 Implementation and Possible Extensions

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 TEST has the effect that purely functional computations are executed almost in the same way as by 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 plan to develop a more serious implementation in C where some extensions will be included. For the evaluation of arithmetic expressions we will provide a direct representation of numbers

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

Figure 7: Code for the example program of subsection 3.2

and an implicit implementation of the arithmetic operations. 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 and a partial list of arguments, and a new instruction APPLY that performs the application of a function represented by a function node to further arguments. 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.

6 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. The project is more advanced than ours, as extensions of the implementation to support lazy evaluation and parallelism have already been developped.

[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 developped 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.

7 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 the functional logic programs that we consider it is possible to recognize deterministic computations by a simple run-time check and thus to treat them in an optimized way.

In addition to the development of a more appropriate implementation it has to be investigated to what extent the narrowing machine can be used to support 'lazy evaluation'. The change of the evaluation strategy seems to be not as straightforward as in reduction machines. Particularly, one has to be careful with backtracking on delayed argument evaluations to avoid nontermination where the innermost strategy would terminate. Another important topic for future work is the exploitation of parallelism and the integration of the narrowing machine in a parallel environment.

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.

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, Springer 1990.

[Lindstrom 87] G.Lindstrom: *Implementing logical variables on a graph reduction architecture*, Workshop on Graph Reduction, LNCS 279, Springer 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 1989.

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

[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 1987, 401–425.

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