New Generation Computing, 11 (1993) 179–215 Ohmsha Ltd./Springer Verlag

Relating the Implementation Techniques of Functional and Functional Logic Languages*

Rita Loogen
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany
e-mail: rita@zeus.informatik.rwth-aachen.de

Abstract

Functional logic languages are declarative programming languages that integrate the programming paradigms of functional and logic languages within a single framework. They are extensions of functional languages with principles derived from logic programming. *Narrowing*, the evaluation mechanism of functional logic languages, can be defined 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, for which we fall back upon the methods of Warren's Prolog engine [Warren 83]. The narrowing machine embodies an optimized treatment of deterministic computations. A complete specification of the reduction and the narrowing machine and of the translation of a sample language into abstract machine code is given. Comparative results of a C-implementation of the reduction and the narrowing machine show that the time overhead of the more complex narrowing evaluation is, in general, less than 10% of the reduction evaluation.

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

*This paper is a revised and extended version of [Loogen 91]

The author has been partially supported by a german-spanish cooperation, funded by the german D.A.A.D and the spanish M.E.C.

[Bellia, Levi 86]). In general, one argues that a logic program has more expressiveness than a comparable functional program. This is due to the more complex execution model of logic languages, SLD-resolution. Functional languages are 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 [Reddy 85,87]. 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 return address 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 also 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. The narrowing machine 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. Of course, the machine can also be seen as an extension of Warren's Prolog engine by features that enable the handling of nested expressions and higher order functions. A special property (non-ambiguity) of the class of functional logic programs that we consider makes an optimized treatment of deterministic computations possible.

The narrowing machine has been implemented in C and compared with an analogous implementation of the reduction machine. It turns out that the time overhead of the narrowing machine is in general less than 10 % of the execution time of the reduction machine. A comparison of the functional and relational evaluation mode shows that functional computations need half the time and half the amount of heap space of logically equivalent relational computations.

The reduction and the narrowing machine, investigated in this paper, realize an eager, i.e. innermost evaluation strategy. The simplicity of this strategy allows us to focus our attention on the new features that are necessary for the transition from reduction machines to narrowing machines. Technically, the implementation of a lazy evaluation strategy causes no problems. However, one has to be careful about the backtracking strategy which induces some nontermination problems. The paper contains a discussion of the implementation of lazy narrowing and the nontermination problems coming along with this strategy. A full treatment of these problems is however out of its scope.

The organisation of the paper is as follows. In Section 2 we describe the syntax, reduction and narrowing semantics of a sample functional and functional logic language, called Simple BABEL. Section 3 introduces a reduction machine for Simple BABEL. The compilation of Simple BABEL programs into machine code for this reduction machine is specified in Section 4. In Section 5, the narrowing machine is developed by an appropriate extension of the reduction machine. Comparative implementation results of both machine models are discussed in Section 6. Section 7 gives an outlook on the implementation of higher order functions and lazy evaluation. A discussion of related work is contained in Section 8. In Section 9, we finally draw some conclusions.

2 A 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, declarative language based on a constructor discipline. Depending on the kind of semantics, reduction or narrowing, Simple BABEL is a functional or a functional logic language, respectively. Consequently, the syntax of Simple BABEL is purely functional.

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. Subsequently, letters $c, d, e \dots$ are used for constructors and the letters $f, g, h \dots$ for function symbols.

The following syntactic domains are distinguished:

- *Variables* $X, Y, Z \dots \in$ *Var*
- *Terms* $s, t, \dots \in$ *Term*: $$\begin{array}{lcll} t & ::= & X & \text{\% Variable} \\ & | & c(t_1, \dots, t_n) & \text{\% } c \in DC^n, n \geq 0 \end{array}$$
- *Expressions* $M, N \dots \in$ *Exp*:

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

$B \rightarrow M$ and $B \rightarrow M_1 \Box M_2$ 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 non-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*: $t_i \in Term$.
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*: If the left hand sides of variable disjoint variants* of any two rules for the same function symbol f:

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

   have a most general unifier σ, then $M\sigma$ and $N\sigma$ are identical†.

The nonambiguity condition guarantees that different defining rules for the same function symbol will not yield contradicting results for the same argument tuple.

The main difference of the syntax of Simple BABEL with respect to the syntax of first order BABEL given in [Moreno, Rodríguez 88] is the *local determinism* condition, which excludes the occurrence of extra or free variables, i.e. variables not occurring on the left hand side, on the right hand side of function rules. In BABEL, the right hand side of a rule may be a guarded expression $B \rightarrow M$, where free variables are allowed in the guard B, but not in the body M. Such guarded rules are necessary for the translation of Horn clause programs into BABEL programs. They are beyond the syntax of functional languages.

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 $n \neq m$

*A *variant* of a rule is obtained by renaming the variables of the rule.

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

Example: In Simple BABEL, the definition of the append function for list concatenation looks like

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

where we use the Prolog notation for list structures.

For more involved BABEL examples, the reader is referred to [Moreno, Rodríguez 89].

2.2 Reduction Semantics

The evaluation mechanism of functional languages (pattern matching and reduction) only allows the evaluation of ground, i.e. variable free, expressions. It is based on the following

2.2.1 Reduction Rules

Let *RExp* denote the set of variable free (reduction) expressions. The reduction rules

$$\longrightarrow \subseteq \mathit{RExp} \times \mathit{RExp}$$

are defined by

1. $(\text{true} \to M) \longrightarrow M \qquad (\text{true} \to M_1 \Box M_2) \longrightarrow M_1 \qquad (\text{false} \to M_1 \Box M_2) \longrightarrow M_2$
2. Let $f(M_1, \ldots, M_m)$ be a ground expression and $f(t_1, \ldots, t_m) := R$ be a rule of some fixed program.

   If $\theta : \mathit{Var} \to \mathit{Exp}$ is a most general unifier with $t_i\theta = M_i$ for $1 \leq i \leq m$ (pattern matching), then:
   $$f(M_1, \ldots, M_m) \longrightarrow R\theta.$$

The second reduction rule describes the application of a program rule for the evaluation of a function application.

2.2.2 Reduction Relation

The nondeterministic *reduction relation* $\Longrightarrow \subseteq \mathit{RExp} \times \mathit{RExp}$ defines the substitution of subexpressions according to the reduction rules. It is defined inductively with respect to the structure of expressions.

- $\longrightarrow \subseteq \Longrightarrow$
- If $M_i \longrightarrow N_i$ for $i \in \{1, \ldots, n\}$ and $\phi \in DC \cup FS$, then
  - $\phi(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow \phi(M_1, \ldots, N_i, \ldots, M_n)$
- $B \longrightarrow B'$ implies
  - $(B \to M) \Longrightarrow (B' \to M) \qquad \text{and} \qquad (B \to M_1 \Box M_2) \Longrightarrow (B' \to M_1 \Box M_2)$

The execution of several reduction steps is described by the transitive, reflexive closure of the reduction relation.

Note that this reduction relation is more restrictive than necessary, because it excludes the simultaneous reduction of several independent subexpressions. We impose this restriction here in order to provide a close relationship of the reduction and the narrowing evaluation mechanism, as the latter does not allow such simultaneous rewriting steps due to the binding of free variables within the evaluated expression.
Reduction of a variable free expression M may lead to the following outcomes:

- *Success:* $M \stackrel{*}{\Longrightarrow} t$ with $t \in \mathit{Term}$,
- *Failure:* $M \stackrel{*}{\Longrightarrow} N$, $N \notin \mathit{Term}$ and no more reduction rule applies to N or to a subexpression of N,
- *Nontermination.*

2.2.3 Reduction Strategy

The nondeterministic reduction relation does not fix the subexpression that has to be reduced first, when a compound expression contains several reducible subexpressions (redexes). The selection of redexes must be specified by a reduction strategy. The most important evaluation schemes for functional languages are the *eager* (leftmost innermost) and the *lazy* (leftmost outermost) reduction strategy. It is well known that the lazy strategy is complete with respect to the nondeterministic reduction relation, i.e. whenever there is a successful reduction of an expression to a term, the lazy evaluation will yield this term, while the eager evaluation strategy may fail to terminate. Nevertheless, we will subsequently focus on the leftmost innermost evaluation strategy, as it keeps the structure of the abstract machine as simple as possible and thus allows us to concentrate on the introduction of the mechanisms for unification and backtracking.

Example: Consider the definition of the list concatenation function *append* at the end of the previous subsection. In functional languages, this definition can be used in a purely functional way to determine the value of applications of *append* to terms or expressions, which do not contain variables, as e.g. $\mathit{append}([a],[b,c])$. The reduction semantics specifies the following evaluation of this expression.

$$\begin{array}{rcl} \mathit{append}([a|[\,]],[b,c]) & \Longrightarrow & [a|\mathit{append}([\,],[b,c])] \\ & \Longrightarrow & [a|[b,c]] = [a,b,c] \end{array}$$

In principle, the reduction mechanism can even handle the evaluation of certain non-ground expressions whose denotation can be expressed by non-ground terms. In the above example, the expression $\mathit{append}([a],L)$ could be evaluated by the reduction rules yielding the term $[a \mid L]$. However, expressions like $\mathit{append}(L,[b,c])$ cannot be evaluated, unless the parameter passing mechanism is generalized as described in the following subsection.

2.3 Narrowing Semantics

Replacing pattern matching by unification generalizes the reduction semantics of Simple BABEL. The generalized evaluation mechanism is called *narrowing.* Unification allows to evaluate non-ground expressions by binding the free variables of the expression to patterns of the program rules and thus to generate a reducible instantiation of the given expression. The result of such a computation sequence is a modified expression together with a substitution for the variables in the original expression. In general, several outcomes are possible for an expression which contains variables.

The formal definition of narrowing is a straightforward generalization of the formal definition of the narrowing semantics.

2.3.1 Narrowing Rules

The narrowing semantics is based on the narrowing rules

$$\aleph\!\!\mapsto_\sigma \subseteq \mathit{Exp} \times \mathit{Exp} \text{ with } \sigma : \mathit{Var} \to \mathit{Exp} :$$

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

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

2. Let $f(M_1, \ldots, M_m)$ be an expression and $f(t_1, \ldots, t_m) := R$ be a variant of a rule of some fixed program, which shares no variables with the expression.

   If $\theta \cup \sigma : \mathit{Var} \to \mathit{Exp}$ is a most general unifier with $t_i\theta = M_i\sigma$ for $1 \leq i \leq m$ (general unification), then:

   $$f(M_1, \ldots, M_m) \aleph\!\!\mapsto_\sigma R\theta.$$

In the second narrowing rule, the simple pattern matching of the corresponding reduction rule has been replaced by the *unification* of the 'pattern' t_i and the expressions M_i, which explicitly allows the binding of variables in M_i. Such bindings are given by the part σ of the most general unifier $\theta \cup \sigma$. θ describes the binding of local variables in the BABEL rule, just as in the reduction semantics. Note that

$$\aleph\!\!\mapsto_\varepsilon \cap (\mathit{Rexp} \times \mathit{RExp}) = \longrightarrow .$$

2.3.2 Narrowing Relation

The *narrowing relation* $\aleph\!\!\Mapsto_\sigma \subseteq \mathit{Exp} \times \mathit{Exp}$ where $\sigma : \mathit{Var} \to \mathit{Exp}$ is inductively defined by:

- $\aleph\!\!\mapsto_\sigma \;\subseteq\; \aleph\!\!\Mapsto_\sigma$
- If $M_i \aleph\!\!\mapsto_\sigma N_i$ for $i \in \{1, \ldots, n\}$ and $\phi \in DC \cup FS$ then
  - $\phi(M_1, \ldots, M_i, \ldots, M_n) \aleph\!\!\Mapsto_\sigma \phi(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
- $B \aleph\!\!\mapsto_\sigma B'$ implies

– $(B \rightarrow M) \aleph\!\!\Rightarrow_\sigma (B' \rightarrow M\sigma)$ and $(B \rightarrow M_1 \square M_2) \aleph\!\!\Rightarrow_\sigma (B' \rightarrow M_1\sigma \square M_2\sigma)$

Again, the execution of several computation steps is given by the transitive, reflexive closure of the narrowing relation, $\aleph\!\!\overset{*}{\Rightarrow}_\sigma$. Additionally, the substitutions of the single steps have to be composed to give the result substitution.
Narrowing of a Simple BABEL expression may lead to success, failure or nontermination, just as reduction. The main difference is the *nondeterminism* contained in the definition of the narrowing rule. Depending on the chosen substitution of free variables, there may be different narrowing derivations of the same function application.

In functional languages, different evaluation sequences are only due to the choice of different redexes. The reduction rules, which specify the rewriting of a single redex, are deterministic. In functional logic languages, additional nondeterminism is present, as there might be several possibilities to narrow a redex.

Example: For the expression $append(L, [b, c])$ the following two narrowing steps are possible:

$$append(L, [b, c]) \begin{cases} \aleph\!\!\Rightarrow_{\{L/[\,]\}} & [b, c] \\ & \\ \aleph\!\!\Rightarrow_{\{L/[X_1|Y_1]\}} & [X_1, append(Y_1, [b, c])], \end{cases}$$

where the recursive call of *append* in the second step has the same shape as the original call. Thus, there are again two alternative subsequent steps.

In the following section we define a reduction machine that implements functional Simple BABEL with an innermost evaluation strategy. In the same way as the narrowing semantics is obtained as a generalization of the reduction semantics, we will later obtain a narrowing machine for Simple BABEL by extending the reduction machine in an appropriate way.

3 A Reduction Machine

In this section we describe an implementation of the reduction semantics of Simple BABEL. For simplicity we consider an innermost evaluation strategy, i.e. the arguments of functions are completely evaluated, before the function application is reduced.

To evaluate a ground expression, the reduction 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 matched with appropriate (leftmost innermost) subexpressions, which are then replaced by the corresponding instances of the rule's right hand sides. The machine tries the program rules in their textual ordering.

The matching of arguments and parameter terms is done explicitly, i.e. without the use of a pattern matching compiler, in order to simplify the later extension for narrowing.

3.1 Components of the Store

The store of the reduction machine contains the following 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.

- *graph* $G : Adr \longrightarrow GNodes_{\mathcal{R}}$
  The graph (heap) is necessary for the representation of terms. In the reduction machine all terms are ground. Thus, the set $GNodes_{\mathcal{R}}$ of graph nodes only contains

  - *constructor nodes:*
    $\langle \text{CONSTR}, c, a_1 : \ldots : a_m \rangle$ with $c \in DC$ and $a_i \in Adr$ $(1 \leq i \leq m)$

  The tag CONSTR is used to be prepared for the later extension of the reduction machine.

  Graph addresses are natural numbers, $Adr := \mathbb{N}$.

- *graph pointer* $gp \in Adr$
  The graph pointer indicates the next "free" position within the graph, $gp = \min\{a \in Adr \mid \forall a' \geq a : \; G(a') \text{ is undefined}\}$. It is used for the allocation of new graph nodes.

- *data stack* $ds \in Adr^*$
  The data stack is used for the graph accesses. Its entries are graph addresses.

  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 the data stack is used during pattern matching to traverse the 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 environments of function calls. During pattern matching one furthermore needs to store some control information, called *choice point*, on top of this stack.

  The *environments* of function calls have the following structure

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

  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 is 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.
- $sep \in \mathbb{N}$ 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.

A *choice point* has the following components

$$\langle tds, n, args,\ badr\rangle,$$

where

- $tds \in \mathbb{N}$ is the top of the data stack minus the number of arguments of the current function call. The arguments are on top of the data stack and will also be saved in the choice point.

  If pattern matching fails, the stack has to be deleted up to this depth and the saved arguments must be copied back onto the stack.
- The n arguments $args \in Adr^*$ of the function call are saved and must be restored on top of the data stack when pattern matching fails and another rule is tried.
- $badr \in PAdr$ is called *backtrack address* and indicates the code address of the next alternative rule.

These simple choice points are necessary in our reduction machine, because pattern matching is done explicitly and not by a pattern matching compiler. The reduction choice points occur only on top of the stack in order to enable the switch to an alternative rule when pattern matching fails. As soon as a matching is successful, i.e. an applicable rule has been found, the choice point is removed from the top of the stack.

The environment stack is accessed via

- the *environment pointer* $ep \in \mathbb{N}$,
  which indicates the topmost environment (function or activation block) on the stack.

  Additionally,
- a *backtrack pointer* $bp \in \mathbb{N}$
  indicates the top of the stack, if there is a choice point on top of it. Otherwise, it is 0.

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

$$(ip, G, gp, ds, st, ep, bp) \in Store_{\mathcal{R}}$$

where

$$Store_{\mathcal{R}} := PAdr \times [Adr \longrightarrow GNodes_{\mathcal{R}}] \times Adr \times Adr^* \times (Adr \cup \mathbb{N} \cup PAdr \cup \{?\})^* \times \mathbb{N}^2.$$

The stacks are assumed to grow to the left. We use the following notations for stacks, graphs and modifications of these components:

- A stack with m elements will be written in the form $el_m : \ldots : el_1$, where el_m is the top element of the stack. If we want to exhibit a finite number k of elements from the rest of the stack, we use the notation $el_1 : \ldots : el_k \cdot st$.

  Let $st = el_m : \ldots : el_1$ be a stack with m elements. Then,

  - $lg(st)$ is called the length of st and denotes the number of elements in the stack, i.e. $lg(st) = m$.
  - $st[i]$ is the element at position i in the stack, i.e. $st[i] = el_i$.
  - $st[i/\tilde{el}]$ denotes the stack st, where the element at position i has been substituted by $\tilde{el}$, i.e.

    $$st[i/\tilde{el}] = el_m : \ldots : el_{i+1} : \tilde{el} : el_{i-1} : \ldots : el_1.$$

  - $st[i..j] := el_i : \ldots : el_j$ denotes the segment of the stack, which contains all elements between the positions i and j, where $m \geq i \geq j \geq 1$.

- Let $G : Adr \longrightarrow GNodes_{\mathcal{R}}$ be a graph, $a_i \in Adr$ and $nd\_i \in GNodes_{\mathcal{R}}$ for $1 \leq i \leq n$.

  Then, $G[a_1/nd_1, \ldots, a_n/nd_n]$ denotes the graph, where the addresses a_i are mapped to the nodes nd_i and all other addresses a are mapped to $G(a)$, i.e.

  $$G[a_1/nd_1, \ldots, a_n/nd_n](a) := \begin{cases} nd_i & \text{if } a = a_i \text{ for some } 1 \leq i \leq n \\ G(a) & \text{otherwise} \end{cases}$$

In the formal specification of the machine instructions of the abstract machines the usage of these notations will be twofold, on the left hand side of the definitions as patterns for the components and on the right hand side as operations for the modification of the components.

3.2 Machine Instructions

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

Matching Instructions

The matching 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 (right to left following the order of the arguments on the stack) and performing local comparisons. As a parameter term may be a variable or a constructor term, we distinguish two instructions for local pattern matching steps. The formal specification of these instructions is given in Figure 1.

- MATCH_VAR 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.

- MATCH_CONSTR (c, n) is used if the parameter term has the top level constructor c and n component terms. 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. The matching of the component terms will be done by the subsequent instructions.

  If the constructors or arities of the pattern and the argument terms are not the same, local backtracking to the next alternative rule is initiated. If no more rule is available, the machine stops. This corresponds to a failing computation. The formal specification of the local backtracking operation is given in Figure 2.

$$
\begin{array}{l}
\mathcal{C}_{\mathcal{R}}\ [\![\text{MATCH\_VAR } i]\!]\ (ip, G, gp, d_0 \cdot ds, st, ep, bp) \\
\quad := (ip+1, G, gp, ds, st[ep-i/d_0], ep, bp) \\
\\
\mathcal{C}_{\mathcal{R}}\ [\![\text{MATCH\_CONSTR } (c,n)]\!]\ (ip, G, gp, d_0 \cdot ds, st, ep, bp) \\
\quad := \mathit{if}\ G(d_0) = \langle \text{CONSTR}, c, b_1 : \ldots : b_n \rangle\ \mathit{then}\ (ip+1, G, gp, b_n : \ldots : b_1 \cdot ds, st, ep, bp) \\
\qquad \mathit{else\ localbacktrack}\ (ip, G, gp, d_0 \cdot ds, st, ep, bp)
\end{array}
$$

Figure 1: Matching Instructions of the Reduction Machine

$$
\begin{array}{l}
\mathit{localbacktrack}(ip, G, gp, d_m : \ldots : d_1, st, ep, bp) \\
\quad := \mathit{if}\ bp = 0\ \mathit{then}\ (0, G, gp, d_m : \ldots : d_1, st, ep, bp) \\
\qquad \mathit{else\ let}\ st = tds : n : a_1 : \ldots : a_n : badr : k : lv_1 : \ldots : lv_k : ep' : ra \cdot st' \\
\qquad\quad \mathit{in}\ (badr, G, gp, a_n : \ldots : a_1 : d_{tds} : \ldots : d_1, \\
\qquad\qquad\qquad tds : n : a_1 : \ldots : a_n : badr : k :? : \ldots :? : ep' : ra \cdot st', ep, bp)
\end{array}
$$

Figure 2: Local Backtracking in the Reduction Machine

$$
\begin{array}{l}
\mathcal{C}_{\mathcal{R}}\ [\![\text{CALL }(f,n,k,j)]\!]\ (ip, G, gp, d_n : \ldots : d_1 \cdot ds, st, ep, bp) \\
\quad := \textit{if } j = 1 \\
\qquad \textit{then}\ \%\ \text{tail recursive call} \\
\qquad\qquad \textit{let } st = k' : lv'_1 : \ldots : lv'_{k'} : ep' : ra' \cdot st' \textit{ in} \\
\qquad\qquad (ca(f), G, gp, d_n : \ldots : d_1 \cdot ds, k : \underbrace{? : \ldots :?}_{k \text{ times}} : ep' : ra' \cdot st', ep - k' + k, bp) \\
\qquad \textit{else}\ \ \textit{let } newenv = k : \underbrace{? : \ldots :?}_{k \text{ times}} : ep : ip + 1 \textit{ in} \\
\qquad\qquad (ca(f), G, gp, d_n : \ldots : d_1 \cdot ds, newenv \cdot st, ep + k + 3, bp),
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{C}_{\mathcal{R}}\ [\![\text{RETURN}]\!]\ (ip, G, gp, ds, k : lv_1 : \ldots : lv_k : ep' : ra \cdot st', ep, bp) \\
\quad := (ra, G, gp, ds, st', ep', bp)
\end{array}
$$

$$
\mathcal{C}_{\mathcal{R}}\ [\![\text{JUMP } l]\!]\ (ip, G, gp, ds, st, ep, bp) := (l, G, gp, ds, st, ep, bp)
$$

$$
\begin{array}{l}
\mathcal{C}_{\mathcal{R}}\ [\![\left\{\begin{array}{c}\text{JUMP\_FALSE}\\ \text{JUMP\_TRUE}\end{array}\right\} l]\!]\ (ip, G[d_0/\langle \textsc{constr}, b, \varepsilon\rangle], gp, d_0 \cdot ds, st, ep, bp) \\
\quad := \textit{if } b = \left\{\begin{array}{c}\text{false}\\ \text{true}\end{array}\right\}\ \textit{then}\ (l, G, gp, ds, st, ep, bp)\ \textit{else}\ (ip + 1, G, gp, d_0 \cdot ds, st, ep, bp)
\end{array}
$$

Figure 3: Instructions for the Forward Control

Forward Control Instructions

The formal specification of these instructions is given in Figure 3.

- The evaluation of function calls is initiated by the instruction CALL (f, n, k, j). The instruction pointer is set to the code address of f, i.e. the first line of code for the function f. A new environment is put on top of the environment stack assuming n pointers to arguments on the data stack, reserving place for k local variables and saving the return address of the function call. The fourth parameter j is a flag that indicates a tail recursive call. If it is set, the previous environment block is overwritten.

- RETURN successfully finishes a function call. The instruction pointer is set to the return address of the environment and the previous environment pointer is restored.

- JUMP l, JUMP_TRUE l, JUMP_FALSE l denote simple and conditional jump instructions.

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{TRY\_ME\_ELSE}\ (l,n)]\!]\ (ip, G, gp, a_n : \ldots : a_1 \cdot ds, st, ep, bp)$$
$$:= (ip+1, G, gp, a_n : \ldots : a_1 \cdot ds, lg(ds) : n : a_1 : \ldots : a_n : l \cdot st, ep, ep+n+3)$$

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{RETRY\_ME\_ELSE}\ l]\!]\ (ip, G, gp, ds, tds : n : a_1 : \ldots : a_n : badr \cdot st, ep, bp)$$
$$:= (ip+1, G, gp, ds, tds : n : a_1 : \ldots : a_n : l \cdot st, ep, bp)$$

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{TRUST\_ME\_ELSE\_FAIL}]\!]\ (ip, G, gp, ds, tds : n : a_1 : \ldots : a_n : badr \cdot st, ep, bp)$$
$$:= (ip+1, G, gp, ds, st, ep, 0)$$

$$\begin{array}{l} \mathcal{C}_{\mathcal{R}}\ [\![\text{DET\_CHECK}]\!]\ (ip, G, gp, ds, st, ep, bp) \\ \quad := \mathit{if}\ bp = 0\ \mathit{then}\ (ip+1, G, gp, ds, st, ep, 0) \\ \qquad\qquad\qquad \mathit{else\ let}\ st = tds : n : a_1 : \ldots : a_n : badr \cdot st' \\ \qquad\qquad\qquad\qquad \mathit{in}\ (ip+1, G, gp, ds, st', ep, 0) \end{array}$$

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{FAIL}]\!]\ (ip, G, gp, ds, st, ep, 0) := (0, G, gp, ds, st, ep, 0)$$

Figure 4: Instructions for the Backward Control

Instructions for the Control of Pattern Matching

The formal specification of the control instructions for pattern matching is given in Figure 4. We use the same instruction names as Warren in his Prolog machine [Warren 83] in order to make clear the close correspondence of the instructions. In the context of the reduction machine, the semantics of these instructions is of course simpler.

- 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, n). A choice point is generated on top of the stack to keep all information necessary to switch to the next alternative rule, whose code starts at program address l. n gives the number of arguments of the function call on top of the stack, which must be saved in the choice point.

- 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.

- DET_CHECK checks whether a choice point is on top of the stack and eliminates it. This instruction is used after a successful pattern matching in order to commit the evaluation of the current function application to the matching rule.

- FAIL leads to an immediate failure stop of the machine. The program address 0 is used as stop address.

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{LOAD } i]\!]\ (ip, G, gp, ds, st, ep, bp) \\ := (ip+1, G, gp, st[ep-i]\cdot ds, st, ep, bp)$$

$$\mathcal{C}_{\mathcal{R}}\ [\![\text{NODE } (c,n)]\!]\ (ip, G, gp, d_n : \ldots : d_1 \cdot ds, st, ep, bp) \\ := (ip+1, G[gp/\langle \text{CONSTR}, c, d_1 : \ldots : d_n\rangle], gp+1, gp\cdot ds, st, ep, bp)$$

Figure 5: Graph Instructions

Graph Instructions

The formal specification of the graph instructions is given in Figure 5.

- LOAD i loads the ith local variable of the current environment on the data stack.
- NODE (c, n) generates a new constructor node at the graph address given by the graph pointer, which is incremented. The n addresses for the components are taken from the data stack and replaced by the address of the newly generated node.

3.3 State Transitions

The transitions of the machine are determined by the code that is generated for a Simple BABEL program. The machine execution starts with the configuration

$$(ca(M), G_0, 0, \varepsilon, \varepsilon, 0, 0),$$

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

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

is applied until one of the following conditions is true.

- $ip = 0$ and $ep = 0$ (successful computation):
  This indicates that the evaluation has been successful. The result is represented by the term whose graph address is on top of the data stack.
- $ip = 0$ and $ep > 0$ (failure):
  In this case a failure has occurred, i.e. some function application could not be evaluated because no rule of the function symbol applied or the guard of some guarded expression could not be evaluated to true.

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:

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

a) Let f be a function symbol with arity m. Then:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{proctrans}\,(\langle f(t_{i1},\ldots,t_{im}) = \mathrm{body}_i \mid 1 \le i \le r\rangle)} \\
\multicolumn{2}{l}{:= \mathit{if}\ r \ge 2\ \mathit{then}} \\
 & \text{TRY\_ME\_ELSE}\ \mathit{rule}_2, m \\
 & \mathit{ruletrans}\ (f(t_{11},\ldots,t_{1m}) = \mathrm{body}_1) \\
\mathit{rule}_2\text{:} & \text{RETRY\_ME\_ELSE}\ \mathit{rule}_3 \\
 & \mathit{ruletrans}\ (f(t_{21},\ldots,t_{2m}) = \mathrm{body}_2) \\
\mathit{rule}_3\text{:} & \vdots \\
 & \vdots \\
\mathit{rule}_r\text{:} & \text{TRUST\_ME\_ELSE\_FAIL} \\
 & \mathit{ruletrans}\ (f(t_{r1},\ldots,t_{rm}) = \mathrm{body}_r) \\
\mathit{else} & \\
 & \mathit{ruletrans}\ (f(t_{11},\ldots,t_{1m}) = \mathrm{body}_1)
\end{array}
$$

b) $\mathit{ruletrans}(f(t_1,\ldots,t_m) = \mathrm{body})$

$$
\begin{array}{rl}
:= & \mathit{matchtrans}\ (t_m) \\
 & \vdots \\
 & \mathit{matchtrans}\ (t_1) \\
 & \text{DET\_CHECK} \\
 & \mathit{exptrans}\ (\text{body},1) \\
 & \text{RETURN}
\end{array}
$$

c) $\mathit{goaltrans}(M)$

$$
\begin{array}{rl}
:= & \mathit{exptrans}\ (M,0) \\
 & \text{JUMP } 0
\end{array}
$$

Figure 6: Compilation Schemes *proctrans*, *ruletrans* and *goaltrans*

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 6a. 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, no choice point will be created.

The translation of each rule consists of code for the matching 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 pattern matching phase the DET_CHECK instruction eliminates an eventual choice point on top of the stack (see Figure 6b). This instruction can of course be omitted in the translation of last or single rules of procedures.

The code for a goal expression is given in Figure 6c. Its translation corresponds to the translation of the right hand side of function rules, followed by an unconditional jump to the stop address 0.

The translation schemes given in Figure 7 are used to produce code for the pattern matching and the evaluation of expressions:

matchtrans : *Term* $\rightarrow$ *Code* generates code, which matches an argument of the actual function call given on top of the data stack with the corresponding term on the left hand side of a rule. As the arguments are in reverse order on top of the stack, matching is done from right to left.

exptrans : *Exp* $\times\ \mathbb{N} \rightarrow$ *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 equals 1, the current environment can be overwritten by the environment of the tail recursive

<table>
<tr>
<td>$matchtrans\ (X_i) :=$ MATCH_VAR i</td>
<td>$matchtrans\ (c(t_1, \ldots, t_n))$
$:=$ MATCH_CONSTR (c, n)
$matchtrans\ (t_n)$
$\vdots$
$matchtrans\ (t_1)$</td>
</tr>
<tr>
<td>$exptrans\ (X_i, j) :=$ LOAD i

$exptrans\ (c(M_1, \ldots, M_n), j)$
$:=$ $exptrans\ (M_1, 0)$
$\vdots$
$exptrans\ (M_n, 0)$
NODE (c, n)

$exptrans\ (f(M_1, \ldots, M_n), j)$
$:=$ $exptrans\ (M_1, 0)$
$\vdots$
$exptrans\ (M_n, 0)$
CALL $(f, n, \sharp vars(f), j)$
where $\sharp vars(f)$ denotes the maximal number of variables in the program rules for f</td>
<td>$exptrans\ (B \rightarrow M, j)$
$:=$ $exptrans\ (B, 0)$
JUMP_TRUE lb_M
FAIL
lb_M: $exptrans\ (M, j)$

$exptrans\ (B \rightarrow M \Box N, j)$
$:=$ $exptrans\ (B, 0)$
JUMP_TRUE lb_T
JUMP_FALSE lb_F
FAIL
lb_T: $exptrans\ (M, j)$
JUMP end_lb
lb_F: $exptrans\ (N, j)$
end_lb: ...</td>
</tr>
</table>

Figure 7: Compilation Schemes *matchtrans* and *exptrans*

call. This is the case for the top level expression on the right hand side of a rule.

Note that the translation scheme given in Figure 7 describes a computation following the leftmost innermost evaluation strategy.

For the small append example program of section 2, the code sequence in Figure 8 will be produced.

5 The Narrowing Machine

In order to do narrowing, the reduction machine must be extended in a way that will be described in this section. In Section 2, we have already seen that the replacement of pattern matching by unification leads to a more powerful evaluation mechanism that allows to evaluate expressions containing variables. These variables become bound during the evaluation and the result of a narrowing sequence is twofold, consisting of the result of the rewriting steps and the substitution of the variables. Depending on the binding of variables, several outcomes may be possible. This inherent nondeterminism of the narrowing semantics is handled by backtracking in the narrowing machine. All possible computation paths are systematically followed. We describe a depth first traversal of the computation tree.

'$append_1$':	TRY_ME_ELSE ($append_2$,2)	MATCH_VAR 2
	MATCH_CONSTR ([ ], 0)	MATCH_VAR 1
	MATCH_VAR 1	MATCH_VAR 3
	DET_CHECK	LOAD 1
	LOAD 1	LOAD 2
	RETURN	LOAD 3
		CALL ($append$,2,3,0)
'$append_2$':	TRUST_ME_ELSE_FAIL	NODE ([. | .],2)
	MATCH_CONSTR ([. | .],2)	RETURN

Figure 8: Code for the *append* Example Program of Section 2

The following extensions are necessary for the reduction machine:

- extension of the graph for the representation of general terms, especially addition of a representation for free variables,
- generalization of the pattern matching instructions to perform unification,
- incorporation of a backtracking facility.

5.1 Components of the Store

The store of the reduction machine is changed in the following way:

- The
  - *program store ps* : $PAdr \dashrightarrow Instr$
  - *instruction pointer ip* $\in PAdr$
  - *graph pointer gp* $\in Adr$
  - *data stack ds* $\in Adr^*$

  remain unchanged.
- The *graph* $G : Adr \dashrightarrow GNodes_{\mathcal{N}}$ of the narrowing machine represents general terms, i.e. variables and structured terms. Additionally, the graph may contain special hole nodes, which will be used during unification for the top down construction of term pattern.

  Thus, the set $GNodes_{\mathcal{N}}$ contains, in addition to the constructor nodes of the reduction machine, 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 and $\langle \text{VAR}, a\rangle$ with $a \in Adr$ embodies a variable that has been bound to the term, whose graph representation starts with address a.
  - *hole nodes:* $\langle \text{HOLE}\rangle$.

- A new component, the *trail* $tr \in Adr^*$, is used to keep track of the bindings of variables in order to be able to undo such bindings on backtracking. It contains the graph addresses of bound variable nodes.
- The *(environment) stack* $st \in (Adr \cup PAdr \cup \mathbb{N} \cup \{?\})^*$ still contains environments and choice points. While the environments remain unchanged, the choice points become more complex.

  In the reduction machine, choice points occur only on top of the stack. After a successful pattern matching, they can be removed, because no other rule needs to be applied. In the narrowing machine, a successful unification does not imply that no other rule is applicable. An alternative computation can be possible using a different binding of free variables. Thus, in general, choice points have to be kept on the environment stack until all rules of a function symbol have been tried. Consequently, the environment stack of the narrowing machine will consist of a mixture of environments and choice points. Each choice point must contain the information that is necessary for resetting the machine to the state, when the corresponding function call was just executed.

  Choice points have the following components

  $$\langle tds, nds, sds, n, args, tt, sgp, sbp, badr \rangle,$$

  where the components $tds, n, args$ and $badr$ are the same as in the choice points of the reduction machine.

  - The first five components
    * $tds \in \mathbb{N}$ ("top of the data stack"),
    * $nds \in \mathbb{N}$ ("number of saved data stack entries"),
    * $sds \in Adr^*$ ("saved data stack entries"),
    * $n \in \mathbb{N}$ (number of arguments) and
    * $args \in Adr^*$ (arguments)

    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. Finally, the n saved arguments $args$ of the function call have to be copied on top of the data stack.
  - $tt \in \mathbb{N}$ indicates the length of the trail to which this must be reset on backtracking. Resetting the trail means unbinding the variable nodes whose addresses are noted in the trail.
  - $sgp \in \mathbb{N}$ saves the graph pointer in order to reset the graph on backtracking.
  - $sbp \in \mathbb{N}$ is the saved backtrack pointer, i.e. the pointer to the previous choice point.
  - $badr \in PAdr$ is the *backtrack address* and indicates the code address of the next alternative rule.

As in the reduction machine, the environment stack is accessed via

- the *environment pointer* $ep \in \mathbb{N}$ which indicates the current environment on the stack and
- the *backtrack pointer* $bp \in \mathbb{N}$ which indicates the topmost choice point.

Note that the current environment is not necessarily the topmost environment on the stack, because there may be choice points saving environments of successfully terminated function calls which are on top of it. In this case, the top of the environment stack is given by the backtrack pointer.

Thus, the state of the narrowing machine will be given by a tuple of the form

$$(ip, G, gp, ds, st, ep, bp, tr) \in Store_{\mathcal{N}}$$

where

$$Store_{\mathcal{N}} := PAdr \times [Adr \longrightarrow GNodes_{\mathcal{N}}] \times Adr \times Adr^{*} \times (Adr \cup \mathbb{N} \cup PAdr \cup \{?\})^{*} \times \mathbb{N}^{2} \times Adr^{*}.$$

To denote the trail component, elements of the trail, trail segments, the length of the trail etc. we use the notation introduced for stacks, except that the trail is assumed to grow to the right.

To sum up, the configuration of the narrowing machine has been developed by extending the 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 bindings and finally
3. by the more complex *choice points* on the environment stack.

5.2 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 arguments of the function call, the length of the trail, the saved graph pointer, 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. 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

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

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. As function g allows for two alternative computations, a choice point is created. The pointer to the argument X is saved in this choice point. At this time, the data stack contains, apart from the argument X, only the representation of a. Consequently, the depth 1 is noted as tds in the choice point. The computation of $g(X)$ using the first rule and binding the variable X to the constructor e is successful and yields a pointer to the graph representation of constructor c on top of the data stack (see Figure 9a). 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 will be eliminated during the unification phase and thus 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 will be destroyed but must be saved for backtracking (see Figure 9b).

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 and the current environment is different from the topmost environment.* 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 minus the number of arguments (see Figure 9a). 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 enable this resetting the number of local variables is the first entry of each environment. The graph pointer is reset to its previous value saved in the choice point. Of course, this indirectly resets the graph. Finally, the data stack has to be reset to the depth tds noted in the choice point, the data stack entries saved in the choice point have to be restored on top of the reduced stack and the saved arguments of the corresponding function call are copied on top of the stack. The formal specification of the backtracking operation is given in Figure 10. The difference with respect to the local backtracking operation of the reduction machine is the resetting of the trail and the more involved restoring of the data stack.

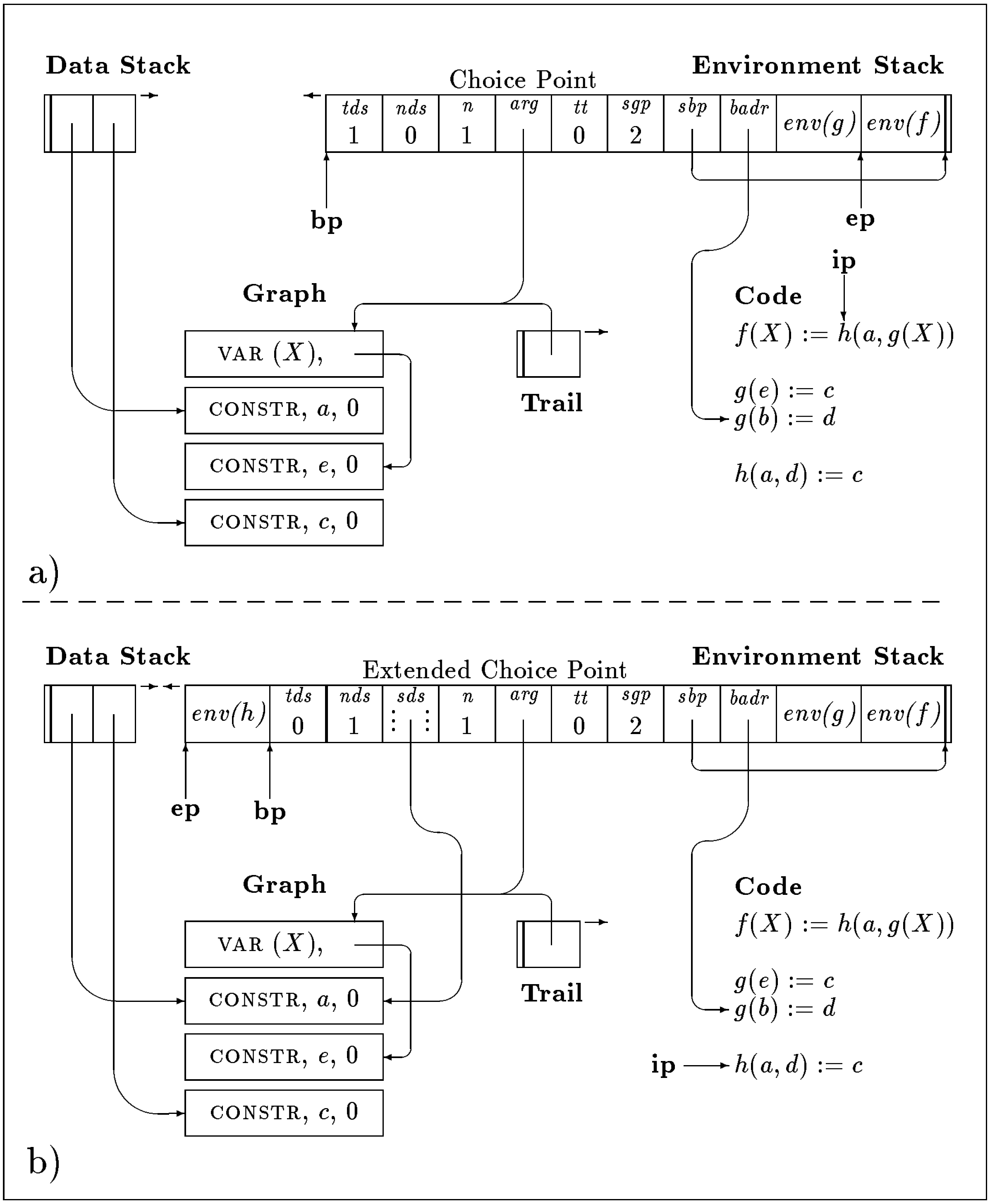

Figure 9: Example — Extension of Choice Points

$$
\begin{array}{l}
backtrack(ip, G, gp, d_n : \ldots : d_1, st, ep, bp, tr) \\
\quad := \mathit{if}\ bp = 0\ \mathit{then}\ (0, G, gp, d_n : \ldots : d_1, st, ep, bp, tr) \\
\qquad \mathit{else\ let}\ st[bp..1] = \\
\qquad\qquad\qquad tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : badr : \\
\qquad\qquad\qquad\qquad\qquad\qquad k : lv_1 : \ldots : lv_k : ep' : ra \cdot st' \\
\qquad\quad \mathit{in}\ (badr, undo(G, tr[(tt+1)..lg(tr)]), sgp, \\
\qquad\qquad a_n : \ldots : a_1 : sd_{nds} : \ldots : sd_1 : d_{tds} : \ldots : d_1, \\
\qquad\qquad tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : badr : \\
\qquad\qquad\qquad\qquad k :? : \ldots :? : ep' : ra \cdot st', bp - n - nds - 7, bp, tr[1..tt]) \\
\\
\text{where } undo(G, tr) := \mathit{if}\ tr = \varepsilon\ \mathit{then}\ G\ \mathit{else\ let}\ tr = tr' \cdot a \\
\qquad\qquad\qquad\qquad\quad \mathit{in}\ undo(G[a/\langle \text{VAR}, ?\rangle], tr')
\end{array}
$$

Figure 10: Backtracking in the Narrowing Machine

5.3 Machine Instructions

The machine instructions of the narrowing machine are the same as the instructions of the reduction machine, but their semantics is extended. In the following, we assume that the reduction semantics of the instructions $\mathcal{C}_{\mathcal{R}}$ is extended for the store of the narrowing machine in such a way, that the further component, the trail, remains unchanged, and describe only the additional cases that must be handled.

In the formal specification of the machine instructions the auxiliary function

$$
\begin{array}{ll}
dereference : [Adr \longrightarrow GNodes_{\mathcal{N}}] \times Adr \longrightarrow Adr & \\
dereference(G, adr) := & \mathit{if}\ G(adr) = \langle \text{VAR}, adr' \rangle \\
& \mathit{then}\ dereference(G, adr')\ \mathit{else}\ adr
\end{array}
$$

is used to unwind chains of bound variable nodes which are handled as reference nodes. In order to avoid dereferencing at each graph access, we always dereference addresses which are written onto the data stack. This implies that a data stack entry never points at a bound variable node.

Matching Instructions

The matching instructions of the reduction machine only allow to bind local variables to arguments of a function call. In the narrowing machine, it is possible that the argument of a function call is a variable that must be bound to a term pattern of the left hand side of a function rule. The construction of the graph representation of term pattern is done node by node, top down right to left, by the sequence of matching instructions that has been generated for the pattern. The newly introduced hole nodes in the graph are used to reserve graph nodes for component structures that have not yet been generated. Thus the two matching instructions of the reduction machine must be extended for the following additional cases:

- MATCH_VAR i generates a new variable node, if the pointer on top of the data stack points at a hole node.

$$
\begin{array}{l}
\mathcal{C}\ [\![\mathrm{MATCH\_VAR}\ i]\!]\ (ip, G, gp, d_0 \cdot ds, st, ep, bp, tr) \\
\quad := \mathit{if}\ G(d_0) = \langle \textsc{hole} \rangle\ \mathit{then}\ (ip+1, G[d_0/\langle \textsc{var}, ? \rangle], ds, st[ep-i/d_0], ep, bp, tr) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \mathcal{C}_{\mathcal{R}}\ [\![\mathrm{MATCH\_VAR}\ i]\!]\ (ip, G, gp, d_0 \cdot ds, st, ep, bp, tr) \\
\\
\mathcal{C}\ [\![\mathrm{MATCH\_CONSTR}\ (c,n)]\!]\ (ip, G, gp, d_0 \cdot ds, st, ep, bp, tr) \\
\quad := \mathit{if}\ G(d_0) = \langle \textsc{var}, ? \rangle \\
\qquad \mathit{then}\ (ip+1, G[d_0/\langle \textsc{var}, gp \rangle, gp/\langle \textsc{constr}, c, gp+1 : \ldots : gp+n \rangle, \\
\qquad\qquad\qquad gp+1/\langle \textsc{hole} \rangle, \ldots, gp+n/\langle \textsc{hole} \rangle], \\
\qquad\qquad\qquad\qquad gp+n+1, gp+n : \ldots : gp+1 \cdot ds, st, ep, bp, tr \cdot d_0) \\
\qquad \mathit{else\ if}\ G(d_0) = \langle \textsc{hole} \rangle \\
\qquad\quad \mathit{then}\ (ip+1, G[d_0/\langle \textsc{constr}, c, gp : \ldots : gp+n-1 \rangle, \\
\qquad\qquad\qquad\quad gp/\langle \textsc{hole} \rangle, \ldots, gp+n-1/\langle \textsc{hole} \rangle], \\
\qquad\qquad\qquad\qquad\qquad gp+n, gp+n-1 : \ldots : gp \cdot ds, st, ep, bp, tr) \\
\qquad\quad \mathit{else}\ \%\ \text{“}\mathcal{C}_{\mathcal{R}}\ [\![\mathrm{MATCH\_CONSTR}\ (c,n)]\!]\ (\ldots)\text{”} \\
\qquad\qquad \mathit{if}\ G(d_0) = \langle \textsc{constr}, c, b_1 : \ldots : b_n \rangle \\
\qquad\qquad \mathit{then}\ \ (ip+1, G, gp, \mathit{dereference}(b_n) : \ldots : \mathit{dereference}(b_1) \cdot ds, st, ep, bp) \\
\qquad\qquad \mathit{else}\ \ \mathit{backtrack}\ (ip, G, gp, d_0 \cdot ds, st, ep, bp)
\end{array}
$$

Figure 11: Unification Instructions

- MATCH_CONSTR (c, n) additionally has to handle the following cases:

  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, hole nodes are constructed. The addresses of these holes are stored in the constructor node and on top of the stack. If the top element of the stack points at a hole node, this node is overwritten by a c-constructor node and again hole nodes are generated for the components.

  The remaining cases are handled as in the reduction machine except that the component addresses, which are copied onto the data stack, are dereferenced.

The formal specification of the extended instructions is given in Figure 11.

Forward Control Instructions

The semantics of the forward control instructions is the same as in the reduction machine except that the CALL instruction may lead to the extension of the top level choice point and the RETURN instruction may only remove the current environment if it is on top of the stack. If a choice point is on top of the stack, the environment must be saved, because it might be needed in an alternative computation.

The formal specification of the extended CALL and RETURN instruction is given in Figure 12.

$$
\begin{array}{l}
\mathcal{C}\ [\![\mathrm{CALL}\ (f,n,k,j)]\!]\ (ip, G, gp, d_n : \ldots : d_1 \cdot ds, st, ep, bp, tr) \\
\quad := \mathit{if}\ bp > ep \\
\qquad \mathit{then\ let}\ newenv = k : \underbrace{? : \ldots : ?}_{k \text{ times}} : ep : ip + 1\ \mathit{in} \\
\qquad\qquad \mathit{if}\ st = tds : nds \cdot st'\ \mathit{and}\ tds > lg(ds)\ \text{\% extension of choice point} \\
\qquad\qquad \mathit{then\ let}\ new = tds - lg(ds)\ \mathit{in} \\
\qquad\qquad\qquad (ca(f), G, gp, d_n : \ldots : d_1 \cdot ds, \\
\qquad\qquad\qquad\ newenv : lg(ds) : nds + new : d_1 : \ldots : d_{new} \cdot st', \\
\qquad\qquad\qquad\qquad\qquad\qquad bp + k + 3 + new, bp + new, tr) \\
\qquad\qquad \mathit{else}\ \ (ca(f), G, gp, ds, newenv \cdot st, bp + k + 3, bp, tr) \\
\qquad \mathit{else}\ \mathcal{C_R}\ [\![\mathrm{CALL}\ (f,n,k,j)]\!]\ (ip, G, gp, d_n : \ldots : d_1 \cdot ds, st, ep, bp, tr) \\
\\
\mathcal{C}\ [\![\mathrm{RETURN}]\!]\ (ip, G, gp, ds, st, ep, bp, tr) \\
\quad := \mathit{let}\ st[ep..1] = k : lv_1 : \ldots : lv_k : ep' : ra \cdot st'\ \mathit{in} \\
\qquad \mathit{if}\ bp > ep\ \mathit{then}\ (ra, G, gp, ds, st, ep', bp, tr) \\
\qquad \mathit{else}\ \mathcal{C_R}\ [\![\mathrm{RETURN}]\!]\ (ip, G, gp, ds, k : lv_1 : \ldots : lv_k : ep' : ra \cdot st', ep, bp, tr)
\end{array}
$$

Figure 12: Extended CALL and RETURN Instructions

Backward Control Instructions

The backward or matching control instructions manage the creation and deletion of choice points on the stack. The only difference with respect to the reduction machine lies in the structure of the choice points. The FAIL instruction, needed for the translation of guarded expressions, now leads to backtracking instead of a failure stop, as there may be alternative computations of the guard expression.

The instruction DET_CHECK is now more involved. In the reduction machine, DET_CHECK is used to remove the choice point on top of the stack, immediately after a successful pattern matching. In the narrowing machine, this is of course not possible, because several rules might be applicable due to 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 DET_CHECK checks 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, locally deterministic computations are recognized and handled as in the reduction machine.
The formal specification of the backward control instructions is given in Figure 13.

Graph Instructions

Due to the presence of bound variable nodes, pointers to the graph are dereferenced when they are copied onto the data stack by the LOAD instruction.

As the NODE instruction takes elements from the data stack, it may be necessary

$$
\begin{array}{l}
\mathcal{C}\ [\![\mathrm{TRY\_ME\_ELSE}\ (l,n)]\!]\ (ip, G, gp, a_n : \ldots : a_1 \cdot ds, st, ep, bp, tr) \\
\quad := (ip+1, G, gp, a_n : \ldots : a_1 \cdot ds, \\
\qquad\qquad\qquad lg(ds) : 0 : n : a_1 : \ldots : a_n : lg(tr) : gp : bp : l \cdot st, ep, ep+n+7, tr) \\
\\
\mathcal{C}\ [\![\mathrm{RETRY\_ME\_ELSE}\ l]\!] \\
\qquad (ip, G, gp, ds, \\
\qquad\quad tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : badr \cdot st, ep, bp, tr) \\
\quad := (ip+1, G, gp, ds, \\
\qquad\qquad tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : l \cdot st, ep, bp, tr) \\
\\
\mathcal{C}\ [\![\mathrm{TRUST\_ME\_ELSE\_FAIL}]\!] \\
\qquad (ip, G, gp, ds, \\
\qquad\quad tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : badr \cdot st, ep, bp, tr) \\
\quad := (ip+1, G, gp, ds, st, ep, sbp, tr) \\
\\
\mathcal{C}\ [\![\mathrm{FAIL}]\!]\ (ip, G, gp, ds, st, ep, bp, tr) := \mathit{backtrack}\ (ip, G, gp, ds, st, ep, bp, tr) \\
\\
\mathcal{C}\ [\![\mathrm{DET\_CHECK}]\!]\ (ip, G, gp, ds, st, ep, bp, tr) \\
\quad := \mathit{if}\ ep > bp\ \mathit{then}\ (ip+1, G, gp, ds, st, ep, bp, tr) \\
\qquad \mathit{else\ let}\ st = tds : nds : sd_1 : \ldots : sd_{nds} : n : a_1 : \ldots : a_n : tt : sgp : sbp : badr \cdot st' \\
\qquad\quad \mathit{in\ if}\ lg(tr) = tt\ \mathit{then}\ (ip+1, G, gp, ds, st', ep, sbp, tr) \\
\qquad\qquad\qquad \mathit{else}\ \ (ip+1, G, gp, ds, st, ep, bp, tr)
\end{array}
$$

Figure 13: Instructions for the Backward Control

to save some of these elements. 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 data stack must be saved in the topmost choice point. This is analogous to the CALL instruction.
The formal specification of the graph instructions is given in Figure 14.

5.4 State Transitions

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

$$(ca(M), G_1, k, \varepsilon, k : \underbrace{0 : \ldots : k-1}_{k \text{ variables}} : 0 : 0, k+3, 0, [\,])$$

where $ca(M)$ denotes the address of the first line of code for M and G_1 is the graph which only contains the unbound variable nodes representing the variables of the goal expression, i.e.

$$G_1(a) = \begin{cases} \langle \mathrm{VAR}, ? \rangle & \text{if } 0 \leq a \leq k-1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Figure 14: Graph Instructions

The environment for the goal expression is necessary to access the variables contained in the goal. The transition rule

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

corresponds exactly to the transition rule of the reduction machine. When the machine stops, we have either

- $ip = 0$ and $ep = k + 3$ (successful computation):
  This indicates that the evaluation has been successful. The result is represented by the top of the data stack while the bindings of the variables of the goal expression are given by the first graph entries which have been reserved for the representation of these variables. If $bp > 0$, more solutions are possible and to obtain these, the machine has to be forced to backtrack.

or

- $ip = 0$ and $bp = 0$ and $ep > k + 3$ (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.

5.5 Code Generation for Narrowing

As the syntax of Simple BABEL is not changed when switching from reduction to narrowing, the code for the narrowing machine is exactly the same code that has been generated for the reduction machine. The generalization of the operational semantics only requires more powerful instructions, as they have been introduced in the previous subsection.

For the small example program of subsection 4.2, the code sequence in Figure 15 will be produced.

This completes the formal description of the narrowing machine for Simple BABEL. In the next section, we present some comparative runtime results of an implementation of the reduction machine and the narrowing machine.

'f':	MATCH_VAR 1	'g':	TRY_ME_ELSE ($rule_2$,1)
	DET_CHECK		MATCH_CONSTR $(e, 0)$
	NODE $(a, 0)$		DET_CHECK
	LOAD 1		NODE $(c, 0)$
	CALL $(g, 1, 0, 0)$		RETURN
	CALL $(h, 2, 0, 1)$	'$rule_2$':	TRUST_ME_ELSE_FAIL
	RETURN		MATCH_CONSTR $(b, 0)$
'h':	MATCH_CONSTR $(d, 0)$		DET_CHECK
	MATCH_CONSTR $(a, 0)$		NODE $(d, 0)$
	DET_CHECK		RETURN
	NODE $(c, 0)$	'$goal$':	LOAD 1
	RETURN		CALL $(f, 1, 1, 0)$
			JUMP 0

Figure 15: Code for the Example Program of Subsection 4.2

6 Implementation Results

The current implementation consists of a compiler frontend that translates BABEL programs into code for the abstract machines, and two interpreter for the abstract machine code, written in C. One interpreter implements the reduction machine while the other one simulates the narrowing machine. The input of the interpreter consists of ASCII files containing the abstract machine code. All examples have been executed on a SPARC station 1+ (Sun-4/65).

Figure 16 compares the execution times for the reduction and the narrowing interpreter for the following example programs which have been written in a functional way.

- *fib* denotes the recursive definition of the fibonacci function.
- *rev* n is the naive definition of the reverse function for lists applied to a list of length n.
- *walk* n means evaluation of the non-naive definition of the reverse function for a list of n elements repeated n times.
- *quicksort* n denotes the recursive quicksort algorithm applied to a list of length n.

The runtimes show that the overhead of the more powerful instructions of the narrowing machine is less than 10% of the execution time. As the precision of the execution times is limited by the clock tick interval, which is 10 msec on SUN-4 systems, a difference in the time behaviour is only measurable for higher execution times. Therefore, we always iterated the applications several times and divided the execution times by the number of iterations. It turned out that the overhead of the iteration could be neglected.

Figure 17 compares the execution times of the narrowing machine interpreter for computations in functional, functional inverted and relational mode for the naive version

example program	reduction	narrowing	difference
fib 10	12,1 msec	12,3 msec	1,7%
fib 15	134,7 msec	136,6 msec	1,4%
fib 18	573,3 msec	579,3 msec	1,1%
reverse 20	23,5 msec	24,4 msec	3,8%
reverse 30	50,3 msec	53,0 msec	5,4%
reverse 40	86,8 msec	91,7 msec	5,6%
walk 50	206,5 msec	220,0 msec	6,5%
walk 100	801,9 msec	862,4 msec	7,5%
quicksort 20	26,3 msec	28,2 msec	7,2%
quicksort 40	58,7 msec	62,7 msec	6,8%

Figure 16: Comparison of Narrowing and Reduction Runtimes

of the list reverse function and a functional program computing one solution of the queens problem.

evaluation mode →		functional	relational	inverted functional	inverted relational
↓ problem	↓ size	'*fun* val = ?'	'*rel*(val,?)'	'*fun* ? = val'	*rel*(?,val)'
reverse	20	24,4 msec	49,1 msec	176,6 msec	304,6 msec
reverse	40	91,7 msec	212,1 msec	1291,0 msec	2322,1 msec
queens	6	154,8 msec	277,8 msec	204,0 msec	371,7 msec
queens	8	989,7 msec	1829,0 msec	1237,7 msec	2319,7 msec
queens	10	1285,0 msec	2426,9 msec	2942,9 msec	5775,3 msec

Figure 17: Comparison of Evaluation Modes

The functional mode shows better execution times, because the purely deterministic evaluation is detected at runtime and thus, the narrowing machine behaves almost as a reduction machine. Rewriting the definition of the functions as binary relations gives the execution time in relational mode. Although, logically, the same evaluation steps are performed, the evaluation needs twice as long, because more unification steps must be executed and the organizational structures, as environments, choice points, the graph and the trail are more extensive. Figure 18 shows the number of graph nodes that are generated for the execution of different calls of the *reverse* function in functional and relational mode.

As one expects evaluations in inverted mode are much more expensive. The proportion of functional and relational computations remains the same.

In this paper, we are mainly interested in the relationship of the reduction and the narrowing evaluation mechanism. For this reason, we omit a comparison of our runtimes with the runtimes given in [Bosco et al. 89] and [Hanus 91]. In general, the runtime of our narrowing interpreter is higher than the runtimes given in these papers. This may

length of list → ↓ evaluation mode	0	5	10
functional	2	32	89
relational	4	94	308

Figure 18: Number of Graph Nodes for Computations of *reverse*

however be due to the different machines and implementation details of the various approaches. In [Bosco et al. 89], the measured programs are C-expanded and executed on a different machine. In [Hanus 91], the abstract machine code is transformed into a compact bytecode, which is then interpreted by a C-emulator.

The next section contains a discussion of the extensions of the narrowing machine that are necessary to implement a more expressive, higher order, lazy functional logic language.

7 Extensions

'Full' BABEL is a higher order, polymorphically typed functional logic language, which uses lazy narrowing as evaluation mechanism. It is more expressive than Simple BABEL, because it allows rules with a guarded expression on the right hand side, where the guard may contain free variables, i.e. variables that do not occur on the left hand side of the rule. This feature allows a simple translation of (pure) Prolog clauses into BABEL rules. The body of a Prolog clause becomes the guard of a BABEL rule, whose body is identical to true. To ensure left linearity the guard must additionally be extended by appropriate equality conditions.
For short, the relationship of Simple BABEL and BABEL can be described by the following equation:

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

The last extension concerns the more general form of the function rules:

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

Note that occurrences of free variables are only allowed in the guard of a rule and not in the body. This restriction is necessary to guarantee determinacy.

According to the more general function rules, the *nonambiguity condition* is generalized in the following way:

Given (variable disjoint variants of) 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].

Considering the above equation the following three extensions are necessary to implement full BABEL on the narrowing machine. Note that polymorphic types have only to be handled by the compiler. During code generation one can assume that the programs are correctly typed.

7.1 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, i.e. higher order variables may occur in the left hand side of rules, but are forbidden to occur free in either right hand side 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, as it is usually done in reduction machines, 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:

$$\begin{array}{lll} \textit{exptrans}\ ((MN), j) := & \textit{exptrans}\ (M, j) & \textit{exptrans}\ (f, j) := \text{FNODE}\ (f, 0) \\ & \textit{exptrans}\ (N, j) & \\ & \text{APPLY} & \end{array}$$

7.2 Lazy Narrowing

BABEL works with a *lazy* narrowing strategy in order to be able to cope with possibly infinite terms. Hence, it tries to narrow expressions at outermost narrowable positions. Narrowing at inner positions is only performed, if it is demanded (by the pattern in the left hand side of some rule) and contributes to some later narrowing step at an outer position.

Unfortunately, the implementation of a lazy narrowing strategy cannot simply be obtained by generalizing the implementation of a lazy reduction strategy. The latter allows to initiate argument evaluations during pattern matching, if such an evaluation is demanded by the pattern of a rule. This is no problem, because such argument evaluations are deterministic. But initiating argument evaluations during unification contains a risk of nontermination, assuming that backtracking is done in the reverse order of forward computations. Let us call this strategy *straightforward lazy narrowing.* Consider the following example program which is due to Juanjo Moreno Navarro:

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

and the goal expression

$$nth(one(X),[0,s(0),s(s(0))]).$$

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 $[0,s(0),s(s(0))]$, 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 preferable 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

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

the following conditions hold:

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

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*

$$\langle \text{SUSPENSION}, \mathit{ca}, \; \mathit{locvars}, \; \mathit{result\ pointer} \rangle,$$

which contain the code address, *ca*, 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 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.

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.

The translation of expressions is changed in such a way that a new translation scheme *delaytrans* is introduced for non-demanded arguments. The necessary modifications of the *exptrans*-scheme and the new scheme are described in Figure 19. Three new instruction are necessary. INITIATE checks whether the address on top of the data stack points at a not yet evaluated suspension node and starts evaluation of this node. SUSPEND generates a suspension node, while the instruction UPDATE writes the result of its evaluation into a suspension node.

A similar treatment of delayed argument evaluations with suspension nodes that are trailed on updates has been described in [Chakravarty, Lock 91].

A remaining problem is the nontermination of the following example that is due to the partially defined function f.

Example: Consider the program

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

exptrans (X_i, j) := LOAD i
INITIATE

exptrans $(c(M_1, \ldots, M_n), j)$
:= *delaytrans* $(M_1, 0)$
$\vdots$
delaytrans $(M_n, 0)$
NODE (c, n)

exptrans $(f(M_1, \ldots, M_n), j)$
:= *exp/delaytrans* $(M_1, 0)$
$\vdots$
exp/delaytrans $(M_n, 0)$
CALL $(f, n, \sharp vars(f), j)$

where *exptrans* is used for demanded arguments and *delaytrans* for non-demanded arguments of f.

delaytrans (X_i, j) := LOAD i

delaytrans $(c(M_1, \ldots, M_n), j)$
:= *delaytrans* $(M_1, 0)$
$\vdots$
delaytrans $(M_n, 0)$
NODE (c, n)

delaytrans (M, j)
:= SUSPEND lb_eval
JUMP lb_c
lb_eval: *exptrans* $(M, 0)$
UPDATE
lb_c: ...

Figure 19: Compilation Schemes for Lazy Evaluation of Uniform Programs

> The lazy evaluation of the goal $f(g(X), a)$ will not terminate, because the evaluation of $g(X)$ does not terminate and following our above definition f demands both parameters. But as the second argument of f is a, no rule is applicable and the evaluation should fail.

The evaluation of demanded arguments of functions in uniform programs is only safe, if the functions are totally defined, i.e. there are rules for each possible combination of argument values. For partially specified functions the evaluation of 'demanded' arguments may cause nontermination instead of a finite failure, as shown in the above example.

7.3 Free Variables in Guards

The translation of the right hand side of guarded rules can be done similar to the translation of guarded expressions. The DET_CHECK 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 rules must be tried if the current rule is applicable and only free guard variables have been bound during unification and evaluation of the guard. The DET_CHECK instruction may then delete all the environments and choice points on top of the current environment. [Loogen, Winkler 91] contains a detailed description of the generalized determinism check and the modifications of the narrowing machine that are necessary to perform this check.

8 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 of the logic plus functional language K-LEAF [Giovannetti et al. 91] 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 in [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 saves space. 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. It is however safe for partially defined functions, because it only forces the evaluation of argument expressions when it is necessary to decide whether a rule is applicable or not.

Our stack narrowing machine incorporates 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 makes an optimized handling of functional computations more difficult. This is one reason for the overhead of relational computations in comparison with analogous functional computations (see Figure 17).

[Hanus 90,91] presents an extension of Warren's Abstract Machine which is able to evaluate Horn clause logic programs with equality. A preprocessor transforms programs into two groups of conditional equations, for narrowing steps and rewriting steps, respectively. Accordingly, the abstract machine is capable of performing narrowing and rewriting, where rewriting is supported by a separate mechanism. Before each narrowing step the machine tries to normalize expressions by rewriting. If no rewriting is possible, narrowing is done. In this way, no narrowing is necessary for the execution of purely functional programs. If however narrowing cannot be avoided, there is an overhead of first trying to perform rewriting. In contrast, our machine always does narrowing, where narrowing is viewed as a generalization of reduction, which is achieved by replacing pattern matching by unification. After the applicability of a rule is proven, it is checked whether pattern matching or unification has been done in order to determine whether rewriting instead of narrowing can be used to evaluate the expression. No overhead is necessary to switch between rewriting and narrowing.

[Chakravarty, Lock 91] presents an amalgamation of the Spineless Tagless G-machine [Peyton-Jones, Salkild 89] and Warren's machine for the implementation of lazy narrowing. It is based on a program transformation, which incorporates the ideas of the transformation introduced in [Moreno et al. 90] and the techniques for the compilation of pattern matching described in [Wadler 87]. Due to the pattern matching compiler, choice points are created for each parameter of a function.

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].

9 Conclusions

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.

The implementation results show that the overhead of the narrowing machine compared to the reduction machine is less than 10%. The interpreter shows reasonable runtimes for the example programs.

Technically, a change from the eager to a lazy evaluation strategy causes no problems. In the same way as in functional reduction machines, it is only necessary to extend the graph of the narrowing machine by suspension nodes for the representation of delayed computations and by new instructions that generate, evaluate and update such suspension nodes. The interaction of delayed computations and backtracking may however lead to nontermination problems, as discussed in Subsection 7.2. The development and implementation of an appropriate lazy evaluation strategy is left for future work.

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 grateful to Stephan Coslar and Stephan Winkler who wrote the C-implementation of the BABEL system that has been used to determine the runtimes given in Section 6. Many thanks are passed to Heiko Vogler, Michael Hanus and Werner Hans who pointed out some inaccuracies in a previous version of the paper and gave valuable comments for its improvement.

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.

[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*, Conference on Logic Programming 1989, MIT Press 1989.

[Chakravarty, Lock 91] M. M. T. Chakravarty, H. C. R. Lock: *The Implementation of Lazy Implementation*, Symposium on Programming Language Implementation and Logic Programming 1991, LNCS 528, Springer Verlag 1991.

[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.

[Giovannetti et al. 91] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: *Kernel LEAF: A Logic plus Functional Language*, Journal of Computer and System Sciences, Vol. 42, No. 2, Academic Press 1991.

[Hanus 90] M. Hanus: *Compiling Logic Programs with Equality*, Workshop on Programming Language Implementation and Logic Programming 1990, LNCS 456, Springer Verlag 1990.

[Hanus 91] M. Hanus: *Efficient Implementation of Narrowing and Rewriting*, International Workshop on Processing Declarative Knowledge, LNCS, Springer Verlag 1991.

[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.

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

[Loogen 91] R. Loogen: *From Reduction Machines to Narrowing Machines*, Colloquium on Combining Paradigms for Software Development — CCPSD, TAPSOFT 1991, LNCS 494, Springer Verlag 1991.

[Loogen, Winkler 91] R. Loogen, St. Winkler: *Dynamic Detection of Determinism in Functional Logic Languages*, Symposium on Programming Language Implementation and Logic Programming 1991, LNCS 528, Springer Verlag 1991.

[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.

[Peyton-Jones, Salkild 89] S. Peyton-Jones, J. Salkild: *The Spineless Tagless G-Machine*, Conference on Functional Programming Languages and Computer Architecture 1989, ACM Press 1989.

[Reddy 85] U. S. Reddy: *Narrowing as the Operational Semantics of Functional Languages*, IEEE Symposium on Logic Programming, IEEE Computer Society Press 1985.

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

[Wadler 87] Ph. Wadler: *Efficient Compilation of Pattern-Matching*, Chapter 5 in: S. Peyton-Jones: *The Implementation of Functional Programming Languages*, Prentice Hall 1987.

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