Theoretical Computer Science 142 (1995) 59–87

Dynamic Detection of Determinism in Functional Logic Languages

Rita Loogen and Stephan Winkler*
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

Abstract

Programs in functional logic languages usually have to satisfy a *nonambiguity condition*, that semantically ensures completeness of conditional narrowing and pragmatically ensures that the defined (non-boolean) functions are deterministic and do not yield different result values for the same argument tuples. The nonambiguity condition allows the dynamic detection of determinism in implementations of functional logic languages. In this paper we show how to achieve this and what can be gained by this optimization.

1 Introduction

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.

In general, functional logic programs have to satisfy a *nonambiguity condition*, that semantically ensures completeness of narrowing [Bosco et al. 89b], [Kaplan 84] and pragmatically ensures that the defined (non-boolean valued) functions are deterministic and do not yield various different result values for the same argument tuples.

The aim of this paper is to show how to make use of the nonambiguity condition in order to perform a *dynamic detection of determinism* in implementations of functional logic languages. The techniques that we present are independent from the implementation method. Especially for the implementation of narrowing, a lot of machine models have been proposed in the literature. A coarse classification distinguishes machines based on the Warren Abstract Machine (WAM) [Warren 83], see e.g. [Balboni et al. 89, Bosco et al. 89a], [Hanus 90], [Mück 90], and extensions of functional reduction machines [Kuchen et al. 90, Moreno et al. 90], [Loogen 91], [Chakravarty, Lock 91], [Wolz 91]. In this paper we focus on the stack narrowing machine of [Loogen 91]. It is however no problem to incorporate a corresponding determinism check in the other machine models.

*The authors are supported by the grant In 20/6–1 from the 'Deutsche Forschungsgemeinschaft'.
e-mail: {rita,winkler}@zeus.informatik.rwth-aachen.de

The effect of the determinism check can be compared to a *dynamic and safe version of the cut operator* known from Prolog. It enables a dynamic reduction of the computation tree, whenever it is safe to do so, due to the nonambiguity of programs. It is important to note that the boolean valued functions corresponding to Prolog predicates always satisfy the nonambiguity condition. Thus, the dynamic detection of determinism applies also to Prolog and logic programs. The main application is however the optimized handling of purely functional computations in functional logic languages.

The paper is organized as follows. In Section 2, we describe the syntax and operational semantics of the functional logic language BABEL [Moreno, Rodríguez 88,92]. This section also contains a discussion of the nonambiguity condition. Section 3 contains an overview of the main components of the stack narrowing machine of [Loogen 91] and the compilation of BABEL programs into code for this machine. We explain the detection of determinism in two stages. In Section 4 the simple case of function definitions without guards, which correspond e.g. to unconditional function definitions and Prolog facts, is handled, while the general case is discussed in Section 5. In both sections we first show under which conditions a rule application is deterministic, i.e. no other rule for the corresponding function symbol must be considered. This is formally proved using the nonambiguity condition. Then we show how to check these conditions in the narrowing machine. Sections 6 and 7 finally contain some conclusions and a discussion of related work, respectively. A complete formal specification of the modified narrowing machine is given in the appendix.

2 The Functional Logic Language BABEL

In this section, we introduce the first order weakly typed subset of the functional logic language BABEL [Moreno, Rodríguez 88,92]. BABEL is a higher order polymorphically typed functional logic language based on a constructor discipline. Its operational semantics is based on narrowing.

2.1 Syntactic Domains

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

The following syntactic domains are distinguished:

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

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

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

2.2 Functional Logic Programs

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

$$\underbrace{f(t_1,\ldots,t_n)}_{\text{lhs}} := \underbrace{\underbrace{\{B \to\}}_{\text{optional guard}} \quad \underbrace{M}_{\text{body}}}_{\text{rhs}} .$$

and satisfy the following conditions:

1. *Term Pattern*: $t_i \in Term$.
2. *Left Linearity*: $f(t_1,\ldots,t_n)$ does not contain multiple variable occurrences.
3. *Restrictions on free variables*: Variables occurring in the right hand side (rhs), but not in the left hand side (lhs), are called *free*. Occurrences of free variables are allowed in the guard, but not in the body.
4. *Nonambiguity*: Given any two rules for the same function symbol f:

   $$f(t_1,\ldots,t_n) := \{B \to\}M. \quad \text{and} \quad f(s_1,\ldots,s_n) := \{C \to\}N.$$

   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 $M\sigma$ and $N\sigma$ are identical. ($M\sigma$ denotes the expression M where all variables X have been replaced by $\sigma(X)$. $M\sigma$ is called an *instance* of M.)

   (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 boolean expressions $B\sigma$ and $C\sigma$ are incompatible.

   In [Moreno, Rodríguez 92] the notion of incompatibility has been formalized in such a way that

   (i) it is decidable whether two expressions B and B' are incompatible, and

   (ii) if B and B' are incompatible first order expressions, there is no Herbrand algebra $\mathcal{A}$ such that, for some substitution ρ over $\mathcal{A}$,

   $$[\![B]\!]_{\mathcal{A}}\rho = [\![B']\!]_{\mathcal{A}}\rho = \text{true}_{\mathcal{A}}.$$

The formal definition of BABEL's declarative semantics is out of the scope of this paper. For our purpose it is only important that the notion of incompatibility of expressions is chosen in such a way that incompatible expressions cannot simultaneously be evaluated to true.

2.1 Example: Let $even(X)$ be true, if X represents an even number and $odd(X)$ be true, if X represents an odd number. Consider the program rules

$$R_1 = \left\{ \begin{array}{lll} f(X) := & even(X) & \to M_1. \\ f(X) := & not(even(X)) & \to M_2. \end{array} \right\}$$

and

$$R_2 = \left\{ \begin{array}{lll} f(X) := & even(X) & \rightarrow M_1. \\ f(X) := & odd(X) & \rightarrow M_2. \end{array} \right\}.$$

Then, the guards of R_1 are incompatible, while the guards of R_2 are not, because incompatibility is only a decidable syntactic approximation of unsatisfiability.

In the next sections we will discuss the consequences of the nonambiguity condition in detail and show how this condition can be used to optimize implementations of functional logic and logic languages by a dynamic detection of deterministic (parts of) computations.

Guarded rules allow a simple translation of Prolog clauses into BABEL. The body of the clause becomes the guard of the BABEL rule whose body is identical to true. To ensure left linearity the guard must be extended by appropriate equality conditions. Note that the nonambiguity condition is trivially satisfied by logic programs, because the bodies of the rules are always identically true and thus condition 4.(b) (fusion of bodies) applies.

2.2 Example:

The Prolog rules

father (bob, john). *male* (X) :- *father* (X,Y).

correspond to the BABEL rules

father (bob, john) := true. *male* (X) := *father* $(X,Y) \rightarrow$ true.

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

- Rules for the boolean operations:

not false	:=	true.
not true	:=	false.

false , Y	:=	false.
true , Y	:=	Y.

false ; Y	:=	Y.
true ; Y	:=	true.

- Rules for the equality operator '=':

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

2.3 Operational Semantics

The operational semantics of BABEL is based on the following *narrowing rule*, which describes how to apply a BABEL rule to an expression through unification:

> Let $f(t_1, \ldots, t_n) := R.$ be some variant of a program rule and $f(M_1, \ldots, M_n)$ an expression which shares no variables with the rule variant.
>
> If $\theta \cup \sigma : Var \rightarrow Exp$ is a most general unifier with $t_i\theta = M_i\sigma$ for $1 \leq i \leq n$, then:
>
> $$f(M_1, \ldots, M_n) \longrightarrow_\sigma R\theta.$$

To evaluate guarded and conditional expressions, one needs the following special narrowing rules:
$(\text{true} \to M) \longrightarrow_\varepsilon M \qquad (\text{true} \to M_1 \Box M_2) \longrightarrow_\varepsilon M_1 \qquad (\text{false} \to M_1 \Box M_2) \longrightarrow_\varepsilon M_2$,
where ε denotes the empty substitution, i.e. $\varepsilon(X) = X$ for all $X \in Var$.

The *one step narrowing relation* $\Longrightarrow_\sigma \subseteq Exp \times Exp$ with $\sigma : Var \to Exp$ is inductively defined by:

- $\longrightarrow_\sigma \subseteq \Longrightarrow_\sigma$
- If $M_i \Longrightarrow_\sigma N_i$ for some $i \in \{1, \ldots, n\}$ and $\phi \in DC \cup FS$, then

$$\phi(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma \phi(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$$

- $B \Longrightarrow_\sigma B'$ implies

$$(B \to M) \Longrightarrow_\sigma (B' \to M\sigma) \quad \text{and} \quad (B \to M_1 \Box M_2) \Longrightarrow_\sigma (B' \to M_1\sigma \Box M_2\sigma)$$

The execution of several computation steps is given by the transitive, reflexive closure of the narrowing relation with composition of the substitutions, $\overset{*}{\Longrightarrow}_\sigma$.

Narrowing of a BABEL expression M may lead to the following outcomes:

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

For simplicity, we consider in the following sections first the *leftmost innermost* narrowing strategy. This corresponds to the evaluation strategy of Prolog. Later on we will discuss the incorporation of our optimization technique in *lazy narrowing* implementations.

2.3 Example: Using the Prolog-like rules of Example 2.2, we get the following innermost narrowing sequence for the expression $male(Z)$.

$$\begin{array}{lll} male(Z) & \Longrightarrow_\varepsilon & (father(Z, Y') \to \text{ true.}) \\ & \Longrightarrow_{\{Z/\text{bob}, Y'/\text{john}\}} & (\text{true } \to \text{ true.}) \\ & \Longrightarrow_\varepsilon & \text{true} \end{array}$$

In [Moreno, Rodríguez 88] soundness and completeness results have been stated, which relate the narrowing semantics to a declarative semantics for first order BABEL. A proof of these results can be found in [Moreno, Rodríguez 92]. The soundness result implies that it is not possible to narrow incompatible expressions simultaneously to true.

2.4 Lemma: Let B and B' be incompatible expressions. If $B \overset{*}{\Longrightarrow}_\sigma$ true for some substitution σ, there exists no narrowing sequence $B'\sigma \overset{*}{\Longrightarrow}_{\sigma'}$ true, where σ' is any substitution.

Proof: The soundness of the narrowing semantics implies that $[\![B\sigma]\!]_\mathcal{A}\rho = \text{true}_\mathcal{A}$ for any Herbrand model $\mathcal{A}$ and any environment ρ. Due to property (ii) of the incompatibility notion we have that $[\![B'\sigma]\!]_\mathcal{A}\rho \neq \text{true}_\mathcal{A}$ for any Herbrand model $\mathcal{A}$ and any environment ρ. Using again the soundness of the narrowing semantics we can deduce that the existence of a narrowing sequence $B'\sigma \overset{*}{\Longrightarrow}_{\sigma'}$ true contradicts this statement. This completes the proof. $\Box$

3 Implementation of Functional Logic Languages

We will explain the dynamic detection of determinism for a simple stack based narrowing machine that has been introduced in [Loogen 91]. The presented optimizing modifications are however independent from the chosen implementation technique and can analogously be incorporated into other machines which are capable of doing narrowing or SLD-resolution. In order to perform backtracking such machines will work with components, which correspond to the trail and the choice points known from Warren's Prolog Engine [Warren 83]. Variables will be represented by special nodes in a heap or graph structure. The modifications that are necessary concern mainly these components. It will be advantageous if the heap is organized in a similar way as a stack as it is e.g. in Warren's Prolog Engine.

In this section we shortly describe the structure and the behaviour of the stack narrowing machine. We concentrate on the components that are involved in the optimizations.

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

3.1 Components of a Narrowing Machine

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

- a *program store* $ps : PAdr \dashrightarrow Instr$ and an

  instruction pointer $ip \in PAdr$, where $PAdr := \mathbb{N}$

The program store contains the translation of the program rules into abstract machine code. This component remains unchanged during the evaluation of programs. The instruction pointer points at the instruction in the program store that has to be executed next.

- a *graph* $G : Adr \dashrightarrow GNodes$, where $Adr := \mathbb{N}$

The graph or heap is necessary for the representation of terms, i.e. variables and structured terms. The set *GNodes* of graph nodes contains the following types of nodes:

- *variable nodes:*

  $\langle \text{VAR}, a\rangle$ with $a \in Adr \cup \{?\}$ ($\langle \text{VAR}, ?\rangle$ represents an unbound variable)

- *constructor nodes:*

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

- a *graph pointer* $gp \in Adr$

The graph pointer always indicates the next free address within the graph. It simplifies the allocation of new graph nodes. The graph is organized as a stack that grows during forward computations and shrinks on backtracking.

- a *data stack* $ds \in Adr^*$

The data stack is used for all accesses to the graph. Its entries are graph addresses.

Instead of a data stack the WAM uses sets of registers. Our modifications, however, do not concern the organisation of the data access.

- an *(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 and *choice points* which keep track of possible alternative computations. Environments contain the control information for forward computations while choice points control backward computations, i.e. backtracking. The environment stack is accessed via two pointers:

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

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 unification the ? will be overwritten by the pointer to the graph node representing the expression to which the local variable is bound.
- $sep \in \mathbb{N}$ is the saved pointer to the previously active environment block.
- $ra \in PAdr$ is the return address of the function call, i.e. the program address at which the computation has to be continued after a successful termination of the function call.

Choice points contain the information that is necessary for restoring a previous state of the machine on backtracking:

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

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

- $tt \in \mathbb{N}$ saves the length of the trail (introduced below).
- $sgp \in \mathbb{N}$ is the saved graph pointer, which is used to reset the graph.
- $sbp \in \mathbb{N}$ 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.

The environment stack of the narrowing machine in general contains a mixture of choice points and environments. The top of the stack is always indicated by the maximum of the environment pointer and the backtrack pointer.

- a *trail* $tr \in Adr^*$

The trail is used to mark variable bindings that have to be undone in case of backtracking.

3.2 Compilation of Functional Logic Programs

After grouping the rules of BABEL programs according to the function symbols, a program has the general form:

$$\mathcal{P} = \{\langle f^{(j)}(t_{i1}^{(j)}, \ldots, t_{in}^{(j)}) = \mathrm{rhs}_i^{(j)}. \mid 1 \leq i \leq r_j \rangle \mid 1 \leq j \leq k\}$$

The code generated for such a program consists of the code for the various procedures (groups of rules for the same function symbol), which will be produced using the scheme *proctrans* given in figure 1. The code for a procedure is executed whenever an application of the corresponding function symbol has to be evaluated. A CALL instruction stores an environment block on top of the environment stack and writes the address of the code for the procedure into the instruction pointer. The defining rules of a function symbol are tested in their textual ordering. Before the first rule is tried, the instruction 'TRY_ME_ELSE (rule_2, n)' puts a choice point on top of the environment stack to keep note of the alternative rules. This choice point saves the current state of the machine. It contains the backtracking address rule_2 and saves the n arguments of the procedure given on top of the stack, because these can be modified during the following unification. If a rule application fails, the state of the machine is restored using the information saved in the choice point, and the next alternative rule is applied. The code for this rule starts with the instruction 'RETRY_ME_ELSE label', which changes the backtrack address within the choice point to the code address of the next alternative rule 'label'. When the last rule is tried, the choice point is removed by the TRUST_ME_ELSE_FAIL instruction.
If there exists only a single rule for the symbol f, the code for the procedure corresponds to the code produced for this rule by the *ruletrans* scheme. The translation of each rule consists of code for the unification of the arguments of the function application with the terms on the left hand side of the rule and code for the evaluation of the right hand side. After evaluation of the guard, the instruction JMP_TRUE tests whether the guard has been evaluated to true and if so, jumps to the code for the evaluation of the body. Otherwise backtracking is initiated by the FAIL command. After a successful evaluation, the RET command gives the control back to the calling procedure. If the environment block of the procedure is on top of the stack, it can be removed by the RET instruction.

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

$$
\begin{array}{lll}
\textit{proctrans}(\langle f(t_{i1},\ldots,t_{in}) := \mathrm{rhs}_i. \mid 1 \le i \le r, r \ge 2\rangle) & & \\
 & := & \text{TRY\_ME\_ELSE } (\textit{rule}_2, n) \\
 & & \textit{ruletrans } (f(t_{11},\ldots,t_{1n}) := \mathrm{rhs}_1.) \\
 & \textit{rule}_2\text{:} & \text{RETRY\_ME\_ELSE } \textit{rule}_3 \\
 & & \textit{ruletrans } (f(t_{21},\ldots,t_{2n}) := \mathrm{rhs}_2.) \\
 & \textit{rule}_3\text{:} & \quad\vdots \\
 & \textit{rule}_r\text{:} & \text{TRUST\_ME\_ELSE\_FAIL} \\
 & & \textit{ruletrans } (f(t_{r1},\ldots,t_{rn}) := \mathrm{rhs}_r.) \\
 & & \\
\textit{ruletrans}(f(t_1,\ldots,t_n) := B \to M.) & := & \textit{unifytrans } (f(t_1,\ldots,t_n)) \\
 & & \textit{exptrans } (\mathrm{B}) \\
 & & \text{JMP\_TRUE body} \\
 & & \text{FAIL} \\
 & \text{body:} & \textit{exptrans } (\mathrm{M}) \\
 & & \text{RET}
\end{array}
$$

Figure 1: Compilation schemes *proctrans* and *ruletrans*

This behaviour corresponds to the stack implementations of ordinary procedure oriented programming languages. If a choice point is on top of the stack, the environment must not be deleted, because it may be needed again in case of backtracking.

For the explanation of our modifications it is not necessary to go further into the details of the translation schemes for the unification, guard and body evaluation and the machine instructions that are used in these schemes. The interested reader is referred to the appendix, which contains a complete formal specification of the innermost narrowing implementation.

4 Determinism Detection for Rules without Guards

In this section we take a closer look at the nonambiguity condition of functional logic programs and show how this condition can be used to detect determinism. We first consider the special case of rules without guards, which is rather common. Most pure function definitions and Prolog facts have this form.

4.1 The Semantics Level

In the case of rules without guards the nonambiguity condition shrinks to the form:

> Given any two rules for the same function symbol f:
>
> $$f(t_1,\ldots,t_n) := M. \text{ and } f(s_1,\ldots,s_n) := N.,$$
>
> if $f(t_1,\ldots,t_n)$ and $f(s_1,\ldots,s_n)$ have a m.g.u. σ, then $M\sigma$ and $N\sigma$ are identical.

and we can state the following simple lemma.

4.1 Lemma: Let

$$(R_1) \;\; f(t_1,\ldots,t_n) := M. \;\text{ and }\; (R_2) \;\; f(s_1,\ldots,s_n) := N.$$

be any two program rules for the same function symbol f and let $f(a_1, \ldots, a_n)$ be an expression with $a_i \in \mathit{Term}$ $(1 \leq i \leq n)$. W.l.o.g. we assume that the rules and the expression do not have common variables.

If $f(t_1, \ldots, t_n)\theta = f(a_1, \ldots, a_n)$ for some substitution θ, i.e. R_1 is applicable without that variables in $f(a_1, \ldots, a_n)$ are bound, then R_2 need not be considered, because either R_2 is not applicable or the result of applying R_2 yields an instance of the application of R_1.

Proof: Assume that also R_2 is applicable, i.e. there exists a substitution θ' with

$$f(s_1, \ldots, s_n)\theta' = f(a_1, \ldots, a_n)\theta'.$$

As R_1 and R_2 do not have common variables and as R_1 is applicable with substitution θ, this implies that

$$f(s_1, \ldots, s_n)\theta\theta' = f(t_1, \ldots, t_n)\theta\theta'.$$

Thus, the left hand sides of R_1 and R_2 are unifiable and the nonambiguity condition tells us that in this case $M\theta\theta' = N\theta\theta'$. As θ binds only variables of R_1, we get $M\theta\theta' = N\theta'$, which shows that $N\theta'$ is an instance of $M\theta$. □

This lemma means that we need not check applicability of other rules whenever no variables in the expression that is being evaluated are bound during the unification with the left hand side of a rule.

The following simple example shows that the application of the redundant rule need not yield the *same* result, but an instance of the result of the application of the first rule.

4.2 Example: Consider the function rules

$$\begin{array}{lcl} f(X, a) & := & X. \\ f(b(a), Y) & := & b(Y). \end{array}$$

The left hand sides of the rules are unifiable, but the nonambiguity condition is satisfied, because the most general unifier of the left hand sides, $\{X/b(a), Y/a\}$, also unifies the right hand sides of the rules.

For the evaluation of the expression $f(b(Z), a)$, both rules are applicable, but only the first rule must be applied, because the global variable Z is not bound during the unification of this expression with the left hand side of the rule. Note that the second rule yields an instance of the result of the first rule:

$$f(b(Z), a) \left\{ \begin{array}{ll} \Rightarrow_{\varepsilon} & b(Z) \\ \Rightarrow_{\{Z/a\}} & b(a) \end{array} \right.$$

4.2 The Implementation Level

The detection of this situation is very simple in our narrowing machine. After a successful unification, we simply have to check whether variables have been bound. Each variable binding is noted in the trail and the length of the trail is noted in the choice point before the unification phase, if there are alternative rules. Thus, we have to compare the length of the trail after the unification with the length noted in the choice point on top of the stack. If the trail has not changed, we are allowed to delete the choice point on top of the environment stack, because Lemma 4.1 ensures that no other rules must be considered for the current function application. We presuppose here that bindings of local rule variables are not noted in the trail. Otherwise, one has to use a flag within the choice point that is set, whenever a global variable is bound.

The specification of the narrowing machine in [Loogen 91] contains already a machine instruction, that checks whether an expression variable has been bound during unification and if so, removes the choice point on top of the stack. In the code for a function with more than one rule this DET_CHECK* instruction is placed after the code for the unification.

$$\begin{array}{rl} \mathit{ruletrans}(f(t_1,\ldots,t_n) := M.) := & \mathit{unifytrans}(f(t_1,\ldots,t_n)) \\ & \text{DET\_CHECK} \\ & \mathit{exptrans}(M) \\ & \text{RET} \end{array}$$

In the code for the last rule of a function and if there is only one rule, the instruction DET_CHECK is superfluous, because the choice point has already been deleted by the instruction TRUST_ME_ELSE_FAIL or no choice point has been created. It should therefore be omitted in these cases.

This special treatment of BABEL functions without guards improves purely or partially deterministic computations. Its effect can be compared to the cut operator that can be used in Prolog to express determinism, because it also cuts off parts of the computation tree. In contrast to the Prolog cut, which has to be placed by the programmer and often restricts the bidirectional use of programs, the instruction DET_CHECK performs a *dynamic cut*, which is *safe*, because no solutions are lost, and which does not restrict the usability of the program.

4.3 Example: The following BABEL rules define a binary function that determines, given a number n and a list l, the prefix of l consisting of n elements.

$$\begin{array}{lcl} \mathit{prefix}(s(N), [Y \mid Ys]) & := & [Y \mid \mathit{prefix}(N, Ys)]. \\ \mathit{prefix}(0, Ys) & := & [\,]. \end{array}$$

Whenever this function is called with ground arguments or with arguments, which have the pattern of the left hand side of the first rule, our optimization applies, i.e. the second rule is not tried. When the first solution is found, one immediately recognizes that no further solutions exist, because there is no more choice point on the stack.

The second advantage of our dynamic detection of determinism is that the rules can nevertheless be used to compute the inverse function or the graph of *prefix*. Expressions of the form $\mathit{prefix}(N, [a, b, c]) = [a, b]$, $\mathit{prefix}(N, Ys) = [a, b, c]$ or $\mathit{prefix}(N, Ys)$ can still be solved appropriately. For the first two goal expressions the representation of the solutions depends on the implementation of the equality rules.

Please note that this detection and optimized treatment of functional computations is not possible in Prolog and logic languages. In these languages functions are represented by predicates, which describe the function graphs. Even in queries which correspond to purely functional calls, i.e. the function arguments are ground terms and the function value is represented by an unbound variable, this result variable will in general be bound during unification and the DET_CHECK instruction would not be applicable.

*In [Loogen 91] the instruction has been called POP, because it eventually pops up a choice point from the environment stack. In this paper, we prefer the more general name DET_CHECK for the same instruction.

4.3 Example (continued): In Prolog the following rules define the same function as in the previous example

$$\begin{array}{l} \mathit{prefix}(s(N),[Y \mid Ys],[Y \mid Zs]) \ \text{:-}\ \mathit{prefix}(N,Ys,Zs). \\ \mathit{prefix}(0,Ys,[\,]). \end{array}$$

The additional third parameter represents the result value. Deterministic calls of these rules have the form $\mathit{prefix}(n,l,Z)$ where n and l are ground terms representing a number and a list, respectively. As the result parameter will be bound during the unification phase, our optimization is not applicable.

The effect of the DET_CHECK instruction could be achieved by placing a cut on the right hand side of the first rule. This would however restrict the applicability of the rules, as e.g., using the rules with the cut, queries of the form $\mathit{prefix}(N,[a,b,c],Z)$ will only yield one solution instead of four solutions without the cut.

In order to optimize the handling of predicates with functional character in Prolog or logic languages, one should distinguish between output, input and general variables using mode declarations and use this information to detect determinism, but the effort seems to be higher than for functional logic languages.

Please note that a determinism check for guardless rules is no longer needed, if a pattern matching compiler, as known from functional languages [Augustsson 85, Peyton Jones 87], or a complete indexing scheme, as known from Prolog and logic languages [Warren 83, Hickey, Mudambi 89], is used. Then, the selection of applicable rules is guided by the structure of the arguments. Choice points are only allocated, when a global variable is bound to some parameter term and alternative bindings are possible. Thus, no choice points are created for the locally deterministic function applications, which are detected by our determinism check (see e.g. [Chakravarty, Lock 91, Wolz 91]).

The simple case of BABEL rules without guards is given in several purely functional definitions. For Prolog clauses, which correspond to boolean valued BABEL functions, this special situation is only present in facts. In the absence of a complete indexing scheme or a pattern driven compilation, our simple determinism check allows in special cases to reduce the computation tree by avoiding unnecessary backtracking and sometimes avoiding the generation of redundant solutions, when there is more than one possibility to derive an (intermediate) solution, which can especially happen in dynamically modified procedures.

In the next section we will consider the more important general case of guarded rules which may contain free variables in the guard. In this general case, the detection of determinism is slightly more involved, but may lead to important reductions of the computation tree especially for logic programs. Its effect cannot be achieved by indexing or intelligent backtracking.

5 Determinism Detection for Rules with Guards

5.1 The Semantics Level

A guarded rule is applicable when the unification of its left hand side with the expression is successful and when the guard of the right hand side of the rule can be evaluated to true. If no variables of the evaluated expression are bound during the unification and during the guard evaluation, no other rule must be tested. Lemma 4.1 can be generalized to the following theorem.

5.1 Theorem: Let

$$(R_1)\;\; f(t_1,\ldots,t_n) := B \to M. \;\text{ and }\; (R_2)\;\; f(s_1,\ldots,s_n) := C \to N.$$

be any two program rules for the same function symbol f and let $f(a_1,\ldots,a_n)$ be an expression with $a_i \in \mathit{Term}$ $(1 \le i \le n)$. W.l.o.g. we assume that the rules and the expression (pairwise) do not have common variables.

If $f(t_1,\ldots,t_n)\theta = f(a_1,\ldots,a_n)$ for some substitution θ and $B\theta \stackrel{*}{\Longrightarrow}_\sigma$ true, where σ binds only local variables of R_1, i.e. R_1 can be applied without that variables in $f(a_1,\ldots,a_n)$ are bound, then R_2 need not be considered, because either R_2 is not applicable or the result of applying R_2 yields an instance of the application of R_1.

Proof: Again we assume that R_2 is also applicable, i.e. there exists a substitution θ' with

$$f(s_1,\ldots,s_n)\theta' = f(a_1,\ldots,a_n)\theta' \text{ and } C\theta' \stackrel{*}{\Longrightarrow}_{\sigma'} \text{ true.}$$

As in the proof of Lemma 4.1 we deduce that $\theta\theta'$ is a unifier of the left hand sides of R_1 and R_2. The general nonambiguity condition tells us now that either

1. $M\theta\theta' = N\theta\theta'$ (fusion of bodies) or
2. $B\theta\theta'$ and $C\theta\theta'$ are incompatible. (Note that instances of incompatible expressions are also incompatible.)

As $B\theta \stackrel{*}{\Longrightarrow}_\sigma \mathit{true}$ and σ binds only free guard variables, the same narrowing steps are possible for an instance of $B\theta$ that binds only non-free variables. Thus, we also have that $B\theta\theta' \stackrel{*}{\Longrightarrow}_\sigma \mathit{true}$. As we have assumed that additionally $C\theta' \stackrel{*}{\Longrightarrow}_{\sigma'}$ true and $C\theta\theta'\sigma = C\theta'$ (θ and σ bind only variables of R_1), Lemma 2.4 implies that $B\theta\theta'$ and $C\theta\theta'$ are *not* incompatible. Consequently, the bodies of R_1 and R_2 must be identical with respect to $\theta\theta'$, i.e. case 1. (fusion of bodies) is true. To complete the proof we now show that the application of R_2 yields an instance of the application of R_1:

R_1 yields the expression $M\theta\sigma$ that equals $M\theta$, because σ binds only free variables of B, which are not allowed to occur in M.

R_2 yields the expression $N\theta'\sigma'$ that equals $N\theta\theta'\sigma'$, because θ does not bind variables in R_2. Because of the fusion bodies, this is identical to $M\theta\theta'\sigma'$ and thus is an instance of $M\theta$. □

This theorem shows that, whenever a rule is applicable without that variables of the expression, which is currently evaluated, are bound, then no other rules for the corresponding function symbol must be considered and no alternative guard evaluations are necessary. Note that Lemma 4.1 is a special case of Theorem 5.1, where the guards of the program rules are identically true.

As Prolog or logic programs are trivially nonambiguous, the theorem is also applicable. Whenever only free variables of the right hand side of a rule have been bound during the application of a rule, no other rule for the corresponding predicate symbol must be considered and no reevaluation of the right hand side of the rule is necessary. This corresponds to a dynamically added cut operator at the rightmost position in a rule.

5.2 The Implementation Level

In order to check whether no other rules must be considered, one has to detect after the unification and a successful guard evaluation, whether only local variables and free guard variables have been bound. A first approach is to place a slightly modified DET_CHECK instruction, which checks whether the length of the trail noted in the choice point corresponding to the active environment block (which is not necessarily on top of the stack, because more environments and choice points may have been pushed on the stack during the guard evaluation) equals the current length of the trail, immediately before the code for the body evaluation. If the check is successful, no global variable has been bound, and all the choice points and environments on top of the active environment can be removed from the stack. Unfortunately, this simple approach applies only for guarded rules without free guard variables.

Guard variables always appear within parameters of predicates (boolean valued functions). When these predicates are called, the unbound (free) local guard variables become global variables, that cannot be distinguished from other global variables. Particularly, their bindings will be trailed during the predicate evaluation. Consequently, the trail will grow in most guard evaluations, even if only free guard variables have been bound.

The main problem is how to detect, whether only free guard variables have been bound during a guard evaluation. For that purpose, the machine has to be able to distinguish between local and global variables, where one has to take into account that the notion of locality is relative with respect to the function calls.

Fortunately, it is, in principal, possible to distinguish between local and global variables of function calls using the graph pointer. Let e.g. sgp be the value of the graph pointer, when a function is called. As global variables (with respect to that function call) are always allocated in the graph before the function call, their addresses are less than sgp. Variable nodes for local variables are allocated after the function call, i.e. their addresses are greater than or equal to sgp.

Thus, the DET_CHECK instruction has simply to check, whether all the addresses, which have been noted in the trail during a unification and guard evaluation, are greater than or equal to the value of the graph pointer at the time of the function call.

A minor additional problem is, however, that the information that is necessary to perform this check, the length of the trail and the graph pointer at the time the function was called, is up to now saved in choice points, which are only generated when a function is defined by more than one rule. The determinism check, however, also applies to functions with only one defining rule, which is guarded and contains free guard variables, because it may allow to remove the choice points and environments of the guard evaluation corresponding to this rule.

5.2 Example: Consider the rules

$$
\begin{array}{lcl}
\mathit{append}([\,], Ys) & := & Ys. \\
\mathit{append}([X \mid Xs], Ys) & := & [X \mid \mathit{append}(Xs, Ys)]. \\
 & & \\
\mathit{sublist}(Xs, Ys) & := & \mathit{append}(L_1, \mathit{append}(Xs, L_2)) = Ys \rightarrow \text{true}
\end{array}
$$

Narrowing of the ground goal expression $\mathit{sublist}([a], [a, b, a, c])$ leads to the following guard evaluation. Note that L_1 and L_2 are free guard variables.

$$
\begin{array}{ll}
\multicolumn{2}{l}{append(L_1, append([a], L_2)) = [a, b, a, c]} \\
\Rightarrow_{\{L_1/[]\}} & append([a], L_2) = [a, b, a, c] \\
\Rightarrow_{\varepsilon} & [a \mid append([\,], L_2)] = [a, b, a, c] \\
\Rightarrow_{\varepsilon} & [a \mid L_2] = [a, b, a, c] \\
\Rightarrow_{\varepsilon} & a = a, L_2 = [b, a, c] \\
\Rightarrow_{\varepsilon} & \text{true}, \ L_2 = [b, a, c] \\
\Rightarrow_{\varepsilon} & L_2 = [b, a, c] \\
\stackrel{*}{\Rightarrow}_{\{L_2/[b,a,c]\}} & \text{true}
\end{array}
$$

Please note that the binding of L_2 is generated using several rules for the predefined function symbols. Only the free guard variables L_1 and L_2 have been bound during this evaluation. Thus the determinism check would be successful, but in the current version of the narrowing machine it is not possible to perform the check, because no choice point for the call of *sublist* has been generated and thus the length of the trail and the graph pointer have not been saved.

In order to save the length of the trail and the graph pointer for each function call, we move this information from the choice point to the environment of a function call. As an environment is allocated for every function call, the determinism check is then always possible.

If the determinism check is successful, Theorem 5.1 guarantees that a reevaluation of the guard is not necessary and alternative function rules need not be considered. This means that all choice points and environments on top of the current environment may be removed from the environment stack. In order to simplify the restoring of the backtrack pointer in this case, we additionally move the saved backtrack pointer from choice points to environments. Thus we perform the following

— *Modification of Environments and Choice Points:*

The structure of the *environments* of function calls becomes

$$\langle nlv, lvars, sep, ra, \underbrace{tt, sgp, sbp}_{\text{new}} \rangle,$$

where the first five components are as before, and the last three components are the additional information, previously stored in the choice point:

- the length of the trail $tt \in \mathbb{N}$ (*t*op of *t*rail)
- the *s*aved *g*raph *p*ointer $sgp \in \mathbb{N}$ and
- the *s*aved *b*acktrack *p*ointer $sbp \in \mathbb{N}$.

Choice points now only contain the following components

$$\langle tds, nds, sds, n, args, badr \rangle.$$

which are necessary to restore a previous machine state on backtracking, in addition to the information that has been moved to the environments.

— *Modification of the Code Generation and the DET_CHECK Instruction*

Now, the following code is produced for a rule.

$$
\begin{array}{rrl}
ruletrans(f(t_1,\ldots,t_n) := B \rightarrow M.) & := & unifytrans\ (f(t_1,\ldots,t_n)) \\
 & & exptrans\ (B) \\
 & & \text{JMP\_TRUE body} \\
 & & \text{FAIL} \\
 & \text{body:} & \text{DET\_CHECK} \\
 & & exptrans\ (M) \\
 & & \text{RET}
\end{array}
$$

The DET_CHECK instruction is placed after the code for the guard evaluation. It checks, whether all addresses, which have been noted in the trail during the unification phase and the guard evaluation, are greater or equal to the graph pointer saved in the current environment block, i.e. correspond to local variables. This check is, of course, only necessary, if there are choice points on top of the current environment, i.e. if the backtracking pointer *bp* is greater than the environment pointer *ep*. If the check is successful, it removes all environments and choice points on top of the actual environment and sets the backtrack pointer to the saved value. If a global variable has been bound, the environment stack remains unchanged. A formal specification of the DET_CHECK instruction is given in Figure 2.

$$
\begin{array}{l}
\mathcal{C}\ [\![\text{DET\_CHECK}]\!]\ (ip, G, gp, ds, st, ep, bp, tr) \\
\quad := \mathit{if}\ ep > bp \\
\qquad \mathit{then}\ (ip+1, G, gp, ds, st, ep, bp, tr) \\
\qquad \mathit{else\ let}\ st[ep..1] = nlv : lvars : sep : ra : tt : sgp : sbp : st' \\
\qquad\quad \mathit{in\ if}\ \forall i\ (tt < i \leq lg(tr)) \rightarrow tr[i] \geq sgp \quad \text{\% Determinism Check} \\
\qquad\qquad \mathit{then}\ (ip+1, G, sgp, ds, st[ep..1], ep, sbp, tr[1..tt]) \\
\qquad\qquad \mathit{else}\ \ (ip+1, G, gp, ds, st, ep, bp, tr)
\end{array}
$$

Figure 2: Formal Specification of the DET_CHECK instruction

The main benefits of our determinism check are the optimized treatment of purely functional computations and the safe and dynamic cut of the computation tree in the general case of nondeterministic computations. By the elimination of choice points, unnecessary backtracking is avoided. Thus, time and space are preserved. An important property of the determinism check is that it often avoids the generation of redundant solutions. This is an advantage of our check in comparison with intelligent backtracking schemes, which also avoid useless backtracking (in general even more than our determinism check), but are not able to detect redundancy.

5.3 Example: Consider again the BABEL rules of Example 5.2.

The goal expression

$$sublist([X,Y],[a,b,a]), sublist([X,Y],[a,b,a,b,a])$$

determines the common two-element sublists of the lists $[a,b,a]$ and $[a,b,a,b,a]$.

Without determinism check, the narrowing machine yields four solutions, where the two solutions $X=a, Y=b$ and $X=b, Y=a$ both appear twice. This redundancy is due to the fact, that the second call of *sublist* is always ground. Only free variables are bound during its evaluation, i.e. backtracking on this call is superfluous.

The determinism check eliminates the choice point of this call and thus avoids the generation of the redundant solutions.

Redundancy often occurs in typical Prolog applications.

5.4 Example: Consider a simple library data base in Prolog:

user (john).	*has_book* (john, b15).	*active* (X) :- *has_book* (X, B).
user (mary).	*has_book* (john, b11).	
user (bob).	*has_book* (john, b5).	
⋮	⋮	
	has_book (mary, b107).	
	⋮	

Without our detection of determinism the query '$user(X), active(X)$' will yield the solution '$X =$ john' as many times as the user john has a book, because the subquery *has_book*(john,B) succeeds several times, binding the free guard variable B to the different books. With our dynamic detection of determinism such redundant solutions are avoided.

Usually, a concrete implementation of the equality operator '=' does not follow the predefined rules given in Section 2.2, but uses a general unification algorithm to test whether two expressions can be made equal or not. Our innermost implementation is based on the narrowing rules

$$t_1 = t_2 \begin{cases} \Longrightarrow_\sigma \ \text{true} & \text{if } \sigma \text{ is m.g.u. of } t_1 \text{ and } t_2 \\ \Longrightarrow_\varepsilon \ \text{false} & \text{otherwise} \end{cases}$$

The main difference lies in the fact that the predefined rules always bind variable parameter to non-variable terms. Narrowing of the expression $X = Y$ yields infinitely many ground solutions, while the above narrowing rules allow to derive the single solution true, binding X to Y, or vice versa. Note that this solution does not represent all the solutions, defined by the predefined rules, as the latter also permit to narrow the expression $X = Y$ to false, binding X and Y to different terms.

The incomplete implementation of the equality operator increases the applicability of the determinism check, as it avoids global variable bindings.

5.5 Example: Again, we refer to the rules of Example 5.2.

Evaluation of the expression $sublist([a], [a, b, a \mid L])$ (cfr. Example 5.2) will not bind the global variable L. Thus, the determinism check applies and only one solution, instead of infinitely many, is generated.

Without the determinism check, it will not be possible to find both solutions of the goal $sublist([X, Y], [a, b, a]), sublist([X, Y], [a, b, a, b \mid L])$ (cfr. the previous example), because there are infinitely many redundant possibilities to solve the second subgoal, given the bindings $\{X/a, Y/b\}$ produced by the first subgoal. Evaluation with the determinism check yields exactly the two possible solutions.

Note that these optimizations cannot be achieved by a dynamic groundness check.

Finally, we want to comment upon the incorporation of the determinism check in lazy narrowing implementations, where the evaluation of arguments in function calls is delayed until their values are demanded by the overall computation.

Usually (see e.g. [Chakravarty, Lock 91] or [Loogen 91]), non-evaluated argument expressions are represented by special *suspension nodes*, which contain the code address for the argument evaluation and the environment that is needed for the execution of this code. Furthermore, these nodes contain place to keep note of the result after a successful evaluation. A suspension node must not be overwritten after its evaluation, because backtracking may require a reset.

Evaluation of a suspension node leads to the creation of a new environment block on top of the stack, execution of the corresponding code 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. Resetting the trail on backtracking unbinds variable and suspension nodes.

Theorem 5.1 has been stated for innermost function applications $f(a_1, \ldots, a_n)$, where $a_i \in \mathit{Term}$, but it also holds for outermost function applications $f(M_1, \ldots, M_n)$ with $M_i \in \mathit{Exp}$, for which the presuppositions are satisfied. The proof remains the same.

Thus, the determinism check is also possible in lazy implementations. As the trail does not only contain the addresses of variable nodes, it must be slightly modified. Whenever the addresses noted in the trail during a successful unification and guard evaluation point at (possibly global) suspensions or local, i.e. free guard variables, the choice points and environments on top of the active environment can be removed.

6 Results and Conclusions

The dynamic detection of determinism has been incorporated in the implementation of our stack narrowing machine. It shows good results, *avoiding unnecessary global backtracking* for deterministic computations after the first result has been found, and in general when local determinism can be detected. The speedup of computations depends of course on the grade of determinism that is present in the various example programs. Table 1 shows the runtimes and stack space for the following example programs, without and with determinism check.

- *colour* n determines all solutions to colour a graph with six nodes using four colours in such a way that connected nodes do not get the same colour. The colours of n nodes are fixed.
- *parse* n parses a simple english sentence of length n.
- *queens* n determines all possibilities to pose n queens on a $n \times n$ chessboard in such a way that they cannot beat each other.
- *quicksort* n sorts the list $[1, \ldots, n]$ using the recursive quicksort algorithm. In the table, only the time to find the solution is noted.

The measurements have been obtained using an interpreter for the abstract machine code of the innermost narrowing machine, written in C. All examples have been executed on a SPARC station 1+ (Sun-4/65).

In most of our examples the runtime could be substantially increased by the determinism check. The time overhead of the additional DET_CHECK instruction is usually less than 3%. Our measurements show an important reduction of the stack space, which is due to the *early deletion of choice points*.

The execution of purely functional programs is the same as in reduction machines. As soon as an applicable rule has been found, the DET_CHECK instruction detects the determinism and deletes the choice point on top of the stack. Note that reduction machines, which do not presume a pattern matching compiler, also have to work with

Problem	*Runtimes (sec)*			*Stack Size (words)*		
	no D.C.	*with D.C.*	*Diff.*	*no D.C.*	*with D.C.*	*Diff.*
colour(0)	1.402	1.124	−19.83 %	147	92	−37.4 %
colour(3)	0.103	0.060	−41.75 %	154	88	−42.9 %
parse(15)	0.100	0.096	−4.00 %	853	363	−57.4 %
queens(8)	7.838	7.928	+1.15 %	1149	697	−39.3 %
quicksort(50)	0.516	0.530	+2.71 %	36555	562	−98.5 %

Table 1: Runtimes and Stack Space without and with Determinism Check (D.C.)

choice points noting the address of the next alternative rule during pattern matching. The only difference to the narrowing machine is the size of choice points and the check that has to be performed to detect the determinism.

The detection of local determinism corresponds to a *dynamic cut* of the computation tree that is *safe*, because it is only done, when no solutions are lost. Note that the modifications of the narrowing machine could also be incorporated in the Warren Abstract Machine [Warren 83].

One could argue that the same or even better backtracking behaviour can be achieved by *intelligent backtracking* schemes. This is true, if backtracking is initialized by a failure. If, however, backtracking is started, because the user asks for more solutions, our method is advantageous, because we delete choice points as soon as determinism is detected. Intelligent backtracking will go back to the topmost choice point, which may be superfluous, while our approach will avoid such redundant computations. Incorporation of an intelligent backtracking scheme in the narrowing machine will of course lead to further improvements in the execution of programs.

7 Related Work

During the last years a lot of proposals have been made for the implementation of narrowing, see e.g. [Balboni et al. 89, Bosco et al. 89a], [Kuchen et al. 90, Moreno et al. 90], [Mück 90], [Hanus 90,91], [Chakravarty, Lock 91], [Wolz 91]. Except of [Kuchen et al. 90, Moreno et al. 90], [Chakravarty, Lock 91] and [Wolz 91], where unification and backtracking have been incorporated in a functional reduction machine, these approaches are based on the Warren Abstract Machine.

In [Loogen 91] it has been shown, how to extend a stack reduction machine for functional languages in order to be able to perform narrowing. This approach led to the question whether it would be possible to detect purely functional computations in the extended machine. This paper gives a positive answer to this question and presents a general method for the dynamic detection of local determinism, that is also useful for purely logic programs.

[Balboni et al. 89, Bosco et al. 89a], [Kuchen et al. 90, Moreno et al. 90] and [Mück 90] do not incorporate a special treatment of deterministic computations.

[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 rewrite expressions. 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 will be 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 proved, it is checked whether pattern matching or unification has been done, i.e. whether reduction or rewriting instead of narrowing can be used to evaluate the expression. No overhead is necessary to switch between rewriting and narrowing.

In [Chakravarty, Lock 91] and [Wolz 91], a pattern driven mechanism is used for the selection of applicable rules. By this method, the determinism in guardless rules is easily detected. For such rules, their approach is even advantageous, because no choice point is created, when it can be avoided. As has been shown in Section 5, the general determinism check is more powerful and leads to further improvements of the backtracking behaviour.
It is straightforward to incorporate the general determinism check in the various abstract machine models.

Acknowledgements

In the previous version of this paper [Loogen, Winkler 91], we proposed another modification of the narrowing machine for the determinism check, which was based on a different graph representation of guard variables.

We are grateful to Lee Naish and Werner Hans, who, independently, pointed out that the introduction of a graph or heap pointer in the narrowing machine, as it is anyway used in Warren's Prolog Machine to shrink the graph on backtracking, simplifies the specification of the determinism check and needs less modifications of the abstract machine. In this revised version of the paper, we followed their proposal.

Furthermore, our work benefitted from discussions with David de Frutos Escrig, Herbert Kuchen, Juanjo Moreno Navarro and Mario Rodríguez Artalejo.

References

[Augustsson 85] L.Augustsson: *Compiling Pattern Matching*, Conf. on Functional Prog. Languages and Computer Arch. 1985, LNCS 201, Springer Verlag 1985.

[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 Progr., Vol.3, 1986, 217–236.

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

[Bosco et al. 89b] P.G.Bosco, C.Cecchi, E.Giovannetti, C.Moiso, C.Palamidessi: *Using resolution for a sound and efficient integration of logic and functional programming*, in: J.de Bakker (ed.): *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, Wiley 1989.

[Chakravarty, Lock 91] M.M.T.Chakravarty, H.C.R.Lock: *The Implementation of Lazy Narrowing*, Symp. on Prog. Language Impl. and Logic Progr. (PLILP) 1991, LNCS 528, Springer Verlag 1991.

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

[Hanus 90] M.Hanus: *Compiling Logic Programs with Equality*, Workshop on Progr. Language Impl. and Logic Progr. (PLILP) 1990, LNCS 456, Springer Verlag 1990.

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

[Hickey, Mudambi 89] T.Hickey, Sh.Mudambi: *Global Compilation of Prolog*, Journal on Logic Progr. 1989, Vol.7, 193–230.

[Kaplan 84] S.Kaplan: *Fair conditional term rewriting systems: unification, termination and confluence*, Technical Report no. 194, University of Orsay 1984.

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

[Loogen 91] R.Loogen: *From Reduction Machines to Narrowing Machines*, Coll. on Comb. Paradigms for Softw. Dev., TAPSOFT 1991, LNCS 494, Springer Verlag 1991.

[Loogen, Winkler 91] R.Loogen, St.Winkler: *Dynamic Detection of Determinism in Functional Logic Languages*, Symp. on Prog. Language Impl. and Logic Progr. (PLILP) 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*, Conf. on Algebraic and Logic Progr. 1988, LNCS 343, Springer Verlag 1989.

[Moreno, Rodríguez 92] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming, Vol. 12, Elsevier Science Publishers 1992.

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

[Mück 90] A.Mück: *Compilation of Narrowing*, Workshop on Progr. Language Impl. and Logic Progr. (PLILP) 1990, LNCS 456, Springer Verlag 1990.

[Peyton Jones 87] S.L.Peyton Jones: *The Implementation of Functional Programming Languages*, Prentice Hall 1987.

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

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

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

[Wolz 91] D.Wolz: *Design of a Compiler for Lazy Pattern Driven Narrowing*, Workshop on the Spec. of Abstract Data Types, LNCS, Springer Verlag 1991.

Appendix

Formal Specification of the Innermost Narrowing Machine with Determinism Check

A Components of the Store

Due to space limitations, we do not repeat the formal specification of the components of the narrowing machine, but refer to Section 3.3 and the modifications described in Section 5.2. Following these definitions, the states of the narrowing machine will be given by tuples of the form

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

where

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

As the program store does not change during the computations of the machine, we do not note it in states. For technical reasons, we extend the set of graph nodes by *hole nodes* $\langle \text{HOLE} \rangle$, which are used during unification for the generation of term representations.

B Backtracking

Backtracking means resetting all components of the machine with respect to the saved information in the top level choice point and the underlying environment.

$$\begin{aligned}
&backtrack(ip, G, gp, d_n : \ldots : d_1, st, ep, bp, tr)\\
&\quad := \mathit{if}\ bp = 0\ \mathit{then}\ (0, G, gp, d_n : \ldots : d_1, st, ep, bp, tr)\\
&\qquad \mathit{else\ let}\ st[bp..1] = tds : nds : sds : n : args : badr :\\
&\qquad\qquad\qquad\qquad k : lv_1 : \ldots : lv_k : ep' : ra : tt : sgp : sbp : st'\\
&\qquad\quad \mathit{in}\ (badr, undo(G, tr[(tt+1)..lg(tr)]), sgp, args : sds : d_{tds} : \ldots : d_1,\\
&\qquad\qquad tds : nds : sds : n : args : badr : k : ? : \ldots : ? : ep' : ra : tt : sgp : sbp : st',\\
&\qquad\qquad\qquad\qquad bp - n - nds - 6, bp, tr[1..tt])
\end{aligned}$$

where $undo(G, tr) := \mathit{if}\ tr = \varepsilon\ \mathit{then}\ G\ \mathit{else\ let}\ tr = a : tr'\ \mathit{in}\ undo(G[a/\langle \text{VAR}, ?\rangle], tr')$
and $lg(tr)$ denotes the length of the trail tr.

C Machine Instructions

Unification Instructions

The unification of an argument of a function call and the formal parameter term of a function rule is done by traversing the parameter term top down (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 unification steps.

UNIFYVAR i binds the ith local variable to the term represented by the top element of the data stack.

$$\begin{aligned}&\mathcal{C}\ [\![\text{UNIFYVAR } i]\!]\ (ip, G, gp, d_0 : ds, st, ep, bp, tr)\\&\quad := \mathit{if}\ G(d_0) = \langle \text{HOLE}\rangle\ \mathit{then}\ (ip+1, G[d_0/\langle \text{VAR}, ?\rangle], gp, ds, st[ep-i/d_0], ep, bp, tr)\\&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ (ip+1, G, gp, ds, st[ep-i/d_0], ep, bp, tr)\end{aligned}$$

UNIFYCONSTR (c, n) is used if the parameter term has the top level constructor c and n component terms. It unifies the root of the term represented by the top element of the data stack with the constructor c. If this local unification fails, backtracking is initiated. Otherwise, the unification of the component terms is done by the subsequent instructions.

$$\begin{aligned}&\mathcal{C}\ [\![\text{UNIFYCONSTR } (c,n)]\!]\ (ip, G, gp, d_0 : ds, st, ep, bp, tr)\\&\quad := \mathit{if}\ G(d_0) = \langle \text{CONSTR}, c, b_1 : \ldots : b_n\rangle\\&\qquad \mathit{then}\ (ip+1, G, gp, \mathit{dereference}(b_n) : \ldots : \mathit{dereference}(b_1) : ds, st, ep, bp, tr)\\&\qquad \mathit{else}\ \ \mathit{if}\ G(d_0) = \langle \text{VAR}, ?\rangle\\&\qquad\quad \mathit{then}\ (ip+1, G[d_0/\langle \text{VAR}, gp\rangle, gp/\langle \text{CONSTR}, c, (gp+1) : \ldots : (gp+n)\rangle,\\&\qquad\qquad\qquad gp+1/\langle \text{HOLE}\rangle, \cdots, gp+n/\langle \text{HOLE}\rangle], gp+n+1,\\&\qquad\qquad\qquad\qquad gp+n : \ldots : gp+1 : ds, st, ep, bp, tr : d_0)\\&\qquad\quad \mathit{else}\ \ \mathit{if}\ G(d_0) = \langle \text{HOLE}\rangle\\&\qquad\qquad \mathit{then}\ (ip+1, G[d_0/\langle \text{CONSTR}, c, gp : \ldots : gp+n-1\rangle,\\&\qquad\qquad\qquad gp/\langle \text{HOLE}\rangle, \cdots, gp+n-1/\langle \text{HOLE}\rangle], gp+n,\\&\qquad\qquad\qquad\qquad gp+n-1 : \ldots : gp : ds, st, ep, bp, tr)\\&\qquad\qquad \mathit{else}\ \ \mathit{backtrack}\ (ip, G, gp, d_0 : ds, st, ep, bp, tr)\end{aligned}$$

where $\mathit{dereference}(G, adr) := \mathit{if}\ G(adr) = \langle \text{VAR}, adr'\rangle\ \mathit{then}\ \mathit{dereference}(G, adr')\ \mathit{else}\ adr$

Forward Control Instructions

The evaluation of new function calls is initiated by the instruction CALL (f, n, k, j). The instruction pointer is set to $ca(f)$, 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.

The CALL instruction may lead to the extension of the top level choice point, if the depth of the stack saved in the choice point on top of the stack is greater or equal to the current depth of the data stack minus the number of arguments of the function, as these arguments will be removed from the data stack during the unification.

$$\begin{aligned}&\mathcal{C}\ [\![\text{CALL } (f,n,k,j)]\!]\ (ip, G, gp, d_n : \ldots : d_1 : ds, st, ep, bp, tr)\\&\quad := \mathit{if}\ bp > ep\\&\qquad \mathit{then}\ \mathit{if}\ st = tds : nds : st'\ \mathit{and}\ tds > lg(ds)\ \text{\% extension of choice point}\\&\qquad\quad \mathit{then}\ \mathit{let}\ n_{\text{save}} = tds - lg(ds)\ \mathit{in}\\&\qquad\qquad (ca(f), G, gp, d_n : \ldots : d_1 : ds,\\&\qquad\qquad\ k : \underbrace{? : \ldots : ?}_{k \text{ times}} : ep : ip+1 : lg(tr) : gp : bp + n_{\text{save}}\\&\qquad\qquad\qquad : lg(ds) : nds + n_{\text{save}} : d_1 : \ldots : d_{n_{\text{save}}} : st',\\&\qquad\qquad\qquad\qquad bp + k + 3 + n_{\text{save}}, bp + n_{\text{save}}, tr)\\&\qquad\quad \mathit{else}\ \ (ca(f), G, gp, ds,\\&\qquad\qquad\qquad k : \underbrace{? : \ldots : ?}_{k \text{ times}} : ep : ip+1 : lg(tr) : gp : bp : st, bp+k+3, bp, tr)\\&\qquad \mathit{else}\ \ \mathit{if}\ j = 1\\&\qquad\quad \mathit{then}\ \text{\% tail recursive call}\\&\qquad\qquad \mathit{let}\ st = k' : a'_1 : \ldots : a'_{k'} : ep' : ra' : tt' : sgp' : bp : st'\ \mathit{in}\\&\qquad\qquad (ca(f), G, gp, d_n : \ldots : d_1 : ds,\\&\qquad\qquad\qquad k : \underbrace{? : \ldots : ?}_{k \text{ times}} : ep' : ra' : lg(tr) : gp : bp : st', ep - k' + k, bp, tr)\end{aligned}$$

$$else\ \ let\ newenv = k : \underbrace{? : \ldots : ?}_{k\ \text{times}} : ep : ip+1 : lg(tr) : gp : bp\ in$$
$$(ca(f), G, gp, d_n : \ldots : d_1 : ds, newenv : st, ep+k+3, bp, tr)$$

RET successfully finishes a function call. It may however 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.

$$\begin{array}{l} \mathcal{C}\ [\![\text{RET}]\!]\ (ip, G, gp, ds, st, ep, bp, tr) \\ \quad := let\ st[ep..1] = k : a_1 : \ldots : a_k : ep' : ra : tt : sgp : bp : st'\ in \\ \qquad if\ bp > ep\ then\ (ra, G, gp, ds, st, ep', bp, tr) \\ \qquad\qquad\qquad else\ (ra, G, gp, ds, st', ep', bp, tr) \end{array}$$

JMP l, JMP_TRUE l, JMP_FALSE l denote simple and conditional jump instructions.

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

$$\begin{array}{l} \mathcal{C}\ [\![\text{JMP\_}\left\{\begin{array}{c}\text{FALSE}\\ \text{TRUE}\end{array}\right\} l]\!]\ (ip, G[d_0/\langle \text{CONSTR}, b, \varepsilon\rangle], gp, d_0 : ds, st, ep, bp, tr) \\ \quad := if\ b = \left\{\begin{array}{c}\text{false}\\ \text{true}\end{array}\right\}\ then\ (l, G, gp, ds, st, ep, bp, tr) \\ \qquad\qquad\qquad\quad else\ \ (ip+1, G, gp, d_0 : ds, st, ep, bp, tr) \end{array}$$

Backward Control Instructions

The backward control instructions manage the creation and deletion of choice points on the stack. 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.

TRY_ME_ELSE (l, n) generates a choice point on top of the stack to keep all information necessary to reset the state of the computation and 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.

$$\begin{array}{l} \mathcal{C}\ [\![\text{TRY\_ME\_ELSE}\ (l, n)]\!]\ (ip, G, gp, d_n : \ldots : d_1 : ds, st, ep, bp, tr) \\ \quad := (ip+1, G, gp, d_n : \ldots : d_1 : ds, lg(ds) : 0 : \varepsilon : n : d_n : \ldots : d_1 : l : st, ep, ep+n+6, tr) \end{array}$$

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.

$$\begin{array}{l} \mathcal{C}\ [\![\text{RETRY\_ME\_ELSE}\ l]\!]\ (ip, G, gp, ds, tds : nds : sds : n : args : badr : st, ep, bp, tr) \\ \quad := (ip+1, G, ds, tds : nds : sds : n : args : l : st, ep, bp, tr) \end{array}$$

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.

$$\begin{array}{l} \mathcal{C}\ [\![\text{TRUST\_ME\_ELSE\_FAIL}]\!]\ (ip, G, gp, ds, tds : nds : sds : n : args : badr : st, ep, bp, tr) \\ \quad := (ip+1, G, gp, ds, st, ep, sbp, tr) \end{array}$$

DET_CHECK performs the determinism check. This instruction has been specified in Figure 2.

FAIL immediately leads to backtracking. It is needed in the translation of guarded expressions to force the reevaluation of the guard, if this has not been evaluated to true.

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

Graph Instructions

LOAD i loads the ith local variable of the current environment on the data stack.

$$
\begin{array}{l}
\mathcal{C}\ [\![\text{LOAD } i]\!]\ (ip, G, gp, ds, st, ep, bp, tr) \\
\quad := \textit{if } st[ep-i] = \,? \\
\qquad \textit{then } (ip+1, G[gp/\langle \textsc{var}, ?\rangle], gp+1, gp : ds, st[ep-i-1/gp], ep, bp, tr) \\
\qquad \textit{else } \ (ip+1, G, gp, \textit{dereference}(G, st[ep-i]) : ds, st, ep, bp, tr)
\end{array}
$$

NODE (c, n) generates a new constructor node where n addresses for the components are taken from the data stack and replaced by the address of the newly generated node. As the NODE instruction takes elements from the data stack, it may be necessary to save some of these elements. This is analogous to the CALL instruction.

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

D State Transitions

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

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

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.

$$G_1(a) = \begin{cases} \langle \textsc{var}, ?\rangle & \text{if } 1 \le a \le k \\ \text{undefined} & \text{otherwise} \end{cases}$$

The environment for the goal expression is necessary to keep track of the bindings of the variables contained in the goal. The transition rule

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

is applied until one of the following conditions is true.

- $ep = 0$ (successful computation):
  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 might be obtained by backtracking.

or

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

E Code Generation

The code that is generated for procedures and rules has already been specified in Section 3.2, where we assume the modifications described in Section 5.1 and 5.2.

The code for a goal expression corresponds to the translation of the right hand side of function rules, followed by an unconditional jump to the stop address 0:

$$\begin{array}{rl} \mathit{goaltrans}(M) := & \mathit{exptrans}\ (M,0) \\ & \mathrm{JMP}\ 0 \end{array}$$

The following translation schemes are used to produce code for the unification:

$$\begin{array}{ll} \mathit{unifytrans}\ (f(t_1,\ldots,t_m)) & \text{with} \\ \quad := \ \mathit{unitrans}\ (t_m) & \\ \qquad \vdots & \\ \qquad \mathit{unitrans}\ (t_1) & \end{array}
\qquad
\begin{array}{ll} \mathit{unitrans}\ (X_i) := \mathrm{UNIFYVAR}\ i & \\ \mathit{unitrans}\ (c(t_1,\ldots,t_n)) := & \mathrm{UNIFYCONSTR}\ (c,n) \\ & \mathit{unitrans}\ (t_n) \\ & \quad \vdots \\ & \mathit{unitrans}\ (t_1) \end{array}$$

and for the evaluation of expressions:

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

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

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

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{exptrans}\ (B \rightarrow M,j)} \\ := & \mathit{exptrans}\ (B,0) \\ & \mathrm{JMP\_TRUE\ lb\_M} \\ & \mathrm{FAIL} \\ \mathrm{lb\_M}: & \mathit{exptrans}\ (M,j) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{exptrans}\ (B \rightarrow M \square N,j)} \\ := & \mathit{exptrans}\ (B,0) \\ & \mathrm{JMP\_TRUE\ lb\_T} \\ & \mathrm{JMP\_FALSE\ lb\_F} \\ & \mathrm{FAIL} \\ \mathrm{lb\_T}: & \mathit{exptrans}\ (M,j) \\ & \mathrm{JMP\ end\_lb} \\ \mathrm{lb\_F}: & \mathit{exptrans}\ (N,j) \\ \mathrm{end\_lb}: & \quad \ldots \end{array}$$

These translation schemes realize the leftmost innermost evaluation strategy.