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
e-mail: {rita,winkler}@zeus.informatik.rwth-aachen.de

Aachener Informatik–Berichte Nr. 91–11

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.

This paper has been published in the proceedings of the 1991 Symposium on Programming Language Implementation and Logic Programming (PLILP), Lecture Notes in Computer Science, Springer Verlag 1991.

1 Introduction

Functional logic languages are extensions of functional languages with principles derived from logic programming [Reddy 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]. 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 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 88,89]. 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 needs to 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.

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 88,89]. 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* $s, t \ldots \in Term$: $$\begin{array}{rcll} t & ::= & X & \text{\% Variable} \\ & | & c(t_1, \ldots, t_n) & \text{\%}\ c \in DC^n, n \geq 0 \end{array}$$
- *Expressions* $M, N \ldots \in Exp$: $$\begin{array}{rcll} M & ::= & X & \text{\% Variable} \\ & | & c(M_1, \ldots, M_n) & \text{\%}\ c \in DC^n, n \geq 0 \\ & | & f(M_1, \ldots, M_n) & \text{\%}\ f \in FS^n, n \geq 0 \\ & | & (B \rightarrow M) & \text{\% guarded expression} \\ & | & (B \rightarrow M_1 \Box M_2) & \text{\% conditional expression} \end{array}$$

$B \rightarrow M$ 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.

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 $f \in FS^m$. Each *defining rule for* f must have the form:

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

and satisfy the following conditions:

1. *Term Pattern*: $t_i \in Term$.
2. *Left Linearity*: $f(t_1, \ldots, t_m)$ 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:

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

one of the three following cases must hold:

(a) *No superposition*: $f(t_1, \dots, t_n)$ and $f(s_1, \dots, s_n)$ are not unifiable.

(b) *Fusion of bodies*: $f(t_1, \dots, t_n)$ and $f(s_1, \dots, s_n)$ 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*: $f(t_1, \dots, t_n)$ and $f(s_1, \dots, s_n)$ have a m.g.u. σ such that the boolean expressions $B\sigma$ and $C\sigma$ are incompatible.

In [Moreno, Rodríguez 89] 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

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

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 $f(t_1, \ldots, t_m) := R$. be some variant of a rule in the program and $f(M_1, \ldots, M_m)$ 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 $1 \leq i \leq m$, then:
>
> $$f(M_1, \ldots, M_m) \longrightarrow_\sigma R\theta.$$

To evaluate guarded and conditional expressions, one needs the following special narrowing rules:

$(\text{true} \rightarrow M) \longrightarrow_\varepsilon M$ $(\text{true} \rightarrow M_1 \square M_2) \longrightarrow_\varepsilon M_1$ $(\text{false} \rightarrow M_1 \square M_2) \longrightarrow_\varepsilon M_2$,

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 $\sigma : Var \rightarrow Exp$ is inductively defined by:

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

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

- $B \longrightarrow_\sigma B'$ implies

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

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

For simplicity we consider in the following sections the *leftmost innermost* narrowing strategy. This corresponds to the evaluation strategy of Prolog. Using the program transformation described in [Moreno et al. 90], our techniques can at once be incorporated in *lazy narrowing* implementations.

2.3 Example: Considering the Prolog-like rules of Example 2.2, we get the following innermost narrowing sequence for the expression $\mathit{male}(X)$.

$$\mathit{male}(X) \quad \Longrightarrow_\varepsilon \quad (\mathit{father}\,(X,Y) \rightarrow \text{true.}) \quad \Longrightarrow_{[X/\text{bob},Y/\text{john}]} \quad (\text{true} \rightarrow \text{true.}) \quad \Longrightarrow_\varepsilon \quad \text{true}$$

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 89]. 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. □

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]. The modifications that are necessary concern mainly these components and the representation of variables in the machine.

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* and an *instruction pointer*
  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.

- *graph* $G : Adr \longrightarrow GNodes$
  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 $a_i \in Adr$ $(1 \leq i \leq m)$

- *(environment) stack*
  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. Thus, environments contain the control information for forward computations while choice points control backward computations, i.e. backtracking. The environment stack is accessed via two pointers:

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

  The *environments* of function calls contain locations for the local variables of the function rules, the arguments of the function call, the saved pointer to the previous environment block and the return address of the function call, i.e. the program address at which the computation has to be continued after a successful termination.

  When the environment block is created, the local variable 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. The arguments are represented by pointers to their graph representation.

Choice points contain the information that is necessary for restoring a previous state of the machine, i.e. the depth of the data stack up to which this has to be deleted on backtracking and, if necessary, a number of saved data stack entries, the length of the trail, the saved backtrack pointer, i.e. the pointer to the previous choice point and finally the backtrack address, which indicates the code address of the next alternative rule. Resetting the trail means unbinding the variables whose addresses have been noted in the trail.

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

- *trail* $tr \in Adr^*$
  The trail is used to mark variable bindings that have to be undone in case of backtracking.

- *data stack* $d \in Adr^*$
  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.

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:

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

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$_2$' puts a choice point 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, which is reset by the 'RETRY_ME_ELSE label' instruction if a rule is not applicable and the next rule is tried. The choice point is removed by the TRUST_ME_ELSE_FAIL instruction, when the last rule is tried.

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.

Let f be a function symbol with arity m and maximally k local variables. Then:

$$
\begin{array}{lrl}
\textit{proctrans}(\langle f(t_{i1},\ldots,t_{im}) := \mathrm{rhs}_i. \mid 1 \leq i \leq r, r \geq 2\rangle) & & \\
 & := & \text{TRY\_ME\_ELSE } \textit{rule}_2 \\
 & & \textit{ruletrans}\ (f t_{11} \ldots t_{1m} := \mathrm{rhs}_1., k) \\
 & \textit{rule}_2: & \text{RETRY\_ME\_ELSE } \textit{rule}_3 \\
 & & \textit{ruletrans}\ (f t_{21} \ldots t_{2m} := \mathrm{rhs}_2., k) \\
 & \textit{rule}_3: & \vdots \\
 & \textit{rule}_r: & \text{TRUST\_ME\_ELSE\_FAIL} \\
 & & \textit{ruletrans}\ (f t_{r1} \ldots t_{rm} := \mathrm{rhs}_r., k) \\
 & & \\
\textit{ruletrans}(f(t_1,\ldots,t_m) := B \rightarrow M., k) & := & \textit{unifytrans}\ (f t_1 \ldots t_m, k) \\
 & & \textit{guardtrans}\ (\mathrm{B}) \\
 & & \text{JMP\_TRUE body} \\
 & & \text{BACKTRACK} \\
 & \text{body:} & \textit{bodytrans}\ (\mathrm{M}, m+k) \\
 & & \text{RET } m+k
\end{array}
$$

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

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 started by the BACKTRACK 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. 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 [Loogen 91].

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 Language 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:

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

if $f(t_1, \ldots, t_m)$ and $f(s_1, \ldots, s_m)$ have a m.g.u. σ, then $M\sigma$ and $N\sigma$ are identical.

and we can state the following simple lemma.

4.1 Lemma: Let $(R_1)\ f(t_1, \ldots, t_n) := M.$ 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. 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 does not need to 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 $N\theta\theta' = N\theta'$, 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.

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 need to be considered for the current function application. Of course we assume here that one does not note bindings of local rule variables 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, called POP, 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 the POP instruction is placed after the code for the unification.

$$\begin{array}{rl} \mathit{ruletrans}(f(t_1,\ldots,t_m) := M., k) := & \mathit{unifytrans}(f(t_1,\ldots,t_m), k) \\ & \mathrm{POP} \\ & \mathit{bodytrans}(M, m+k) \\ & \mathrm{RET}\ m+k \end{array}$$

In the code for the last rule of a function and if there is only one rule, the POP instruction is superfluous, because the choice point has already been deleted by the 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 POP performs a *dynamic cut*, which is *safe*, because no solutions are lost, and which does not restrict the usability of the program.

4.2 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 expression the representation of the solutions depends on the implementation of the equality rules, which is out of the scope of this paper.

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 POP instruction would not be applicable.

4.2 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 directly applicable.

The effect of the POP instruction can be achieved by placing a cut on the right hand side of the first rule. This will 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 could 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.

In most Prolog systems the above example will be handled in an optimized way by the use of *indexing* that has for this example a similar effect as our POP instruction. In general, indexing is however only used for the first argument. Swapping the first two arguments of *prefix* makes indexing useless, but does not affect the dynamic detection of determinism in BABEL.

The simple case of BABEL rules without guards is given in most purely functional definitions. For Prolog clauses, which correspond to boolean valued BABEL functions, this special situation is only present in facts. Nevertheless, our detection of determinism reduces in special cases 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 general case of guarded rules which may contain free variables in the guard. In the general case the detection of determinism is slightly more involved, but may lead to important reductions of the computation tree especially for logic programs.

5 Determinism Detection for Rules with Guards

5.1 The Language 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 during the unification and during the guard evaluation no variables of the evaluated expression are bound, no other rule needs to be tested. Lemma 4.1 can be generalized to the following theorem.

5.1 Theorem: Let (R_1) $f(t_1, \ldots, t_n) := B \rightarrow M.$ and (R_2) $f(s_1, \ldots, s_n) := C \rightarrow N.$ be any two program rules for the same function symbol f and let $f(a_1, \ldots, a_n)$ be an expression with $a_i \in Term$ $(1 \leq i \leq n)$. W.l.o.g. 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 and $B\theta \stackrel{*}{\Longrightarrow}_\sigma$ true, where σ binds only free variables in B, i.e. R_1 is applicable without that variables in $f(a_1, \ldots, a_n)$ are bound, then R_2 does not need to 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 $B\theta \stackrel{*}{\Longrightarrow}_\sigma true$ and σ binds only free guard variables, the same narrowing steps are possible for an instance of B that binds only non-free variables. Thus, we also have that $B\theta\theta' \stackrel{*}{\Longrightarrow}_\sigma true$. 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 need to be considered and no alternative guard evaluations are necessary.

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 function symbol needs to 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 need to be considered, one has to detect whether after the unification and a successful guard evaluation only local variables and free guard variables have been bound. A first approach is to place a POP instruction immediately before the code for the body evaluation, 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. If this is the case, 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.

When a guard contains free variables, the detection whether only the free guard variables have been bound during the evaluation of the guard is not possible in the current version of the narrowing machine. The guard variables always appear within parameters of predicates. When these predicates are called, the 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.
Consequently, the following modifications are necessary in our narrowing machine:

— *Distinct Graph Representation of Guard Variables*
In order to be able to check whether only guard variables have been bound, one has to distinguish the representation of variables and guard variables in the machine.

Furthermore, nested guard evaluations make it necessary to distinguish the free guard variables of different guards. This is achieved by adding a pointer to the environment of the function call that initiated the guard evaluation, to graph representation of the guard variable.

Thus, we have chosen to represent the guard variables by the following type of nodes:

$$\langle \text{GUARDVAR}, a, ep\rangle \text{ with } a \in Adr \cup \{?\} \text{ and } ep \in \mathbb{N}.$$

Such guard variables are generated by a newly introduced instruction

'INIT_GUARD_VARS nr',

which is placed immediately before the code for the guard evaluation. The parameter is the number of free guard variables within the environment of the function call.

— *Extension of Environments*
The detection of determinism also makes sense for functions with only one defining rule, because it may allow to remove the choice points and environments of the guard evaluation corresponding to this rule. In order to be able to handle such cases, where no choice point is generated that saves the length of the trail, we have to note the length of the trail additionally in the environment block of a function call.

— *Extension of the Trail and the POP Instruction*
To simplify the test whether the variables noted in the trail after a guard evaluation are guard variables or not, we note additional information on the bound variables in the trail. We define $tr \in ((\text{VAR} \times Adr) \cup (\text{GUARDVAR} \times Adr \times \mathbb{N}))^*$.

The tags VAR/GUARDVAR indicate the type of the bound variable and for guard variables we additionally note the environment pointer that is stored in the node of the guard variable.

The extended trail allows a simple check whether only guard variables, whose corresponding environment block lies above or is equal to the block of the current function call, i.e. the saved environment pointer is greater or equal to the current environment pointer, have been bound during a guard evaluation. Note that guard variables of previous function calls (saved environment pointer is less than the actual environment pointer) have to be considered as global variables. The POP instruction is modified appropriately. It removes all environments and choice points on top of the actual environment, when the check of the trail is successful. If a global variable has been bound, the environment stack remains unchanged.

Consequently, the following code is produced for a rule. The additional parameter of *ruletrans* gives the number of free guard variables within the rule. We assume that the guard variables are placed after the other local variables within the environment block of the function call.

$ruletrans(f(t_1, \ldots, t_m) := B \rightarrow M., k, gv)$	$:=$	$unifytrans\ (ft_1 \ldots t_m, k)$
		INIT_GUARD_VARS gv
		$guardtrans$ (B)
		JMP_TRUE body
		BACKTRACK
	body:	POP
		$bodytrans$ (M,$m+k$)
		RET $m+k$

We conclude this section considering a small example.

5.2 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.

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. In most of our examples the runtime could be increased by 10 to 40 % of the runtime without the determinism check. The additional test that is performed by the POP instruction does not slow down computations too much. The *early deletion of choice points* increases the space efficiency of the machine. [Winkler 91] contains a detailed discussion of examples and run time results.

The execution of purely functional programs is the same as in reduction machines. As soon as an applicable rule has been found, the POP 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 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 superfluous 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]. Except of [Kuchen et al. 90, Moreno et al. 90] and [Chakravarty, Lock 91], where unification and backtracking have been incorporated in a functional reduction machine, all 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 can also be useful for purely logic programs.

Except of [Hanus 90,91] the other proposals do not contain a special treatment of deterministic computations, but it seems that it is straightforward to integrate our techniques into the various machine models.

[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.

References

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

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

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

[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 Prog. (PLILP) 1991, LNCS, 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 Programming Language Implementation and Logic Programming (PLILP) 1990, LNCS 456, Springer Verlag 1990.

[Hanus 91] M.Hanus: *Efficient Implementation of Narrowing and Rewriting*, Draft Paper, University of Dortmund 1991.

[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 Prog. (ESOP) 1990, LNCS 432, Springer Verlag 1990.

[Loogen 91] R.Loogen: *From Reduction Machines to Narrowing Machines*, Coll. on Combining Paradigms for Software Development, TAPSOFT 1991, LNCS 494, 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*, Conf. on Algebraic and Logic Prog. (ALP) 1990, LNCS 463, Springer Verlag 1990.

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

[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.

[Winkler 91] St. Winkler: *Development of a Program System for the Functional Logic Language BABEL by Implementing a Program Environment, a Compiler and a Stack Based Abstract Machine*, Diploma Thesis, RWTH Aachen, June 1991 (in german).