On the Interaction of Lazy Evaluation and Backtracking

*Werner Hans, Rita Loogen and Stephan Winkler**

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

Aachener Informatik-Berichte Nr. 92–12

Abstract

We investigate the interaction of lazy evaluation and backtracking in the framework of functional logic languages, whose operational semantics is based on lazy narrowing. Technically, it is no problem to realize a lazy narrowing strategy by adapting the well-known techniques, which have been developed for functional languages, to the more general evaluation mechanism of functional logic languages. But, unfortunately, it turns out, that the use of a lazy strategy has some severe disadvantages. In particular, it may lead to nontermination in combination with backtracking, where an innermost strategy will determine a solution. The use of demandedness information for function arguments allows us to define a mixture between an eager and a lazy evaluation strategy, which partially helps to cope with these problems. The runtimes obtained for various example programs with respect to the different strategies, substantiate that the mixed strategy is a reasonable compromise between an eager and a lazy strategy for functional logic languages.

This paper has been published in the proceedings of the 1992 Symposium on Programming Language Implementation and Logic Programming (PLILP), Lecture Notes in Computer Science, Springer Verlag 1992.

*The authors are supported by the grant In 20/6–1 from the 'Deutsche Forschungsgemeinschaft'.
(e-mail address for correspondence: rita@informatik.rwth-aachen.de)

1 Introduction

Lazy evaluation is the evaluation mechanism of modern functional languages as e.g. Miranda [17] and Haskell [9]. It delays the evaluation of function arguments, until their values are definitely needed to determine the result of the function application. Thus, lazy evaluation avoids superfluous computations. Furthermore, it enables the use of possibly infinite data structures.

Backtracking is used in the implementation of logic languages to handle the rule selection nondeterminism. In general, it realizes a depth first search of the computation tree of a query.

In the framework of purely functional or purely logic languages the two mechanisms do not come into contact, as there is no need for arbitrary backtracking in functional languages and there are no nested predicate applications in logic languages.

During the last decade, there has however been an increasing interest in the integration of the functional and logic programming paradigms within a uniform language framework [1], [5]. Coarsely, one can distinguish two main research directions within this area. On the one hand, Horn clause logic languages have been enhanced with equality and functions (see e.g. [7], [8]), where one essentially assumes an innermost evaluation strategy for nested function applications. On the other hand, functional languages have been augmented with logical capabilities (see e.g. [2], [13]. The resulting languages are usually characterized as functional logic languages [16], which retain functional syntax but use narrowing instead of reduction as operational semantics. Narrowing is a generalization of reduction, which is obtained by using unification instead of pattern matching for parameter passing. Functional logic languages usually focus on a lazy evaluation strategy. This is the context, where lazy evaluation meets backtracking.

The aim of our paper is to show that it is in principle no problem to incorporate a lazy evaluation strategy into a narrowing implementation based on an abstract machine, whose backtracking mechanism has been borrowed from the well-known Warren Abstract Machine (WAM) [18], by using the techniques which have been developed for functional languages [6], [15]. But, unfortunately, the combination of lazy evaluation and backtracking has some negative effects. The delayed evaluation of arguments causes backtracking to initiate the re-evaluation of arguments before switching to another rule for the function symbol of the application. This implies on the one hand that arguments may be repeatedly evaluated for several function rules. On the other hand, the evaluation of a function application will not terminate, whenever an argument yields infinitely many solutions which do not match with the pattern of the currently selected function rule.

This risk of nontermination is not present in implementations that combine an eager (innermost) evaluation strategy and backtracking. With an innermost evaluation strategy all rules of a function are tried before backtracking on the arguments. Because there are only finitely many rules for each function symbol, nontermination may not occur during rule selection.

Multiple re-evaluations of locally deterministic arguments can simply be avoided by refining the backtracking mechanism. The nontermination problem is however more obstinate. It is only possible to avoid divergence during rule selection by evaluating arguments before the execution of function calls, which is however not safe, if the program admits nonterminating derivation sequences. For this reason, we will consider demandedness (strictness) information for function arguments to guarantee safeness and to get rid of the problems. The demandedness information allows us to define a mixture of an eager and a lazy evaluation strategy, which allows the eager evaluation of demanded arguments. It will however turn out, that this approach avoids the nontermination problems only for a subclass of programs, which will be syntactically characterized. An example program, for which nontermination due to backtracking can only be avoided by the eager evaluation of a non-demanded function argument, shows that it is impossible to find a strategy that is safe and avoids the nontermination problems of lazy narrowing plus backtracking, in general. The mixed strategy shows acceptable runtimes in comparison with an eager and a lazy strategy.

The paper is organized as follows. In Sect. 2, we present a very simple subset of a functional logic language. Section 3 describes our lazy narrowing implementation. In Sect. 4, the disadvantages of the lazy narrowing implementation are discussed. Section 5 shows in how far demandedness information helps to cope with the nontermination problems. Section 6 compares the runtimes of the innermost, the lazy and the mixed strategy for several example programs. Section 7 contains a discussion of related work and the conclusions.

2 A Sample Functional Logic Language

Our sample language Mini-BABEL is a first order subset of the functional logic language BABEL [13], which will be sufficient to discuss the interaction of lazy evaluation and backtracking in the framework of a functional logic language. BABEL, itself, is a higher order polymorphically typed functional logic language based on a constructor discipline. Its operational semantics is lazy narrowing.

2.1 Syntactic Domains

Let $\langle S, DC, FS\rangle$ be a first order signature with the set S of base types, the alphabet $DC = \bigcup_{(w,s)\in S^*\times S} DC^{(w,s)}$ of typed ***constructors*** and the alphabet $FS = \bigcup_{(w,s)\in S^*\times S} FS^{(w,s)}$ of typed ***function symbols***.

In the following, letters $c, d, e \ldots$ are used for constructors and letters $f, g, h \ldots$ for function symbols. In our example programs, we usually assume that S contains the type ***nat*** and that DC contains the constructors $0 \in DC^{(\varepsilon,nat)}$ and $s \in DC^{(nat,nat)}$. The following syntactic domains are distinguished:

- *Variables* $X, Y, Z \ldots \in Var = \bigcup_{s\in S} Var^s$
- *Terms* $s, t \ldots \in Term = \bigcup_{s\in S} Term^s$:

t	::=	X	% Variable
	$\mid$	$c(t_1,\ldots,t_n)$	% $c \in DC^{(s_1\ldots s_n,s)}$, $t_i \in Term^{s_i}, n \geq 0$

- *Expressions* $M, N \ldots \in Exp = \bigcup_{s\in S} Exp^s$:

M	::=	X	% Variable
	$\mid$	$\varphi(M_1,\ldots,M_n)$	% $\varphi \in DC^{(s_1\ldots s_n,s)} \cup FS^{(s_1\ldots s_n,s)}$,
			% $M_i \in Exp^{s_i}, n \geq 0$

2.2 Mini-BABEL Programs

A ***Mini-BABEL program*** consists of a finite set of defining rules for the function symbols in FS. Let $f \in FS^{(s_1\ldots s_n,s)}$. Each ***defining rule for*** f must have the form:

$$f(t_1,\ldots,t_n) := M.$$

and satisfy the following conditions:

1. *Term Pattern and Well Typedness*: $t_i \in Term^{s_i}$ $(1 \leq i \leq n)$ and $M \in Exp^s$.
2. *Left Linearity*: $f(t_1,\ldots,t_n)$ does not contain multiple variable occurrences.
3. *Local Determinism*: Variables occurring in M must also occur in $f(t_1,\ldots,t_n)$.
4. *Nonambiguity*: Given any two (variable disjoint variants) of rules for the same function symbol f:

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

   such that $f(t_1,\ldots,t_n)$ and $f(s_1,\ldots,s_n)$ have a unifier σ, then $M\sigma$ and $N\sigma$ must be identical. ($M\sigma$ denotes the expression M where all variables X have been replaced by $\sigma(X)$.)

f is ***totally defined***, if for all $\tilde{t}_j \in Term^{s_j}$ $(1 \leq j \leq n)$ there exists (a variable disjoint variant of) a defining rule $f(t_1,\ldots,t_n) := M$ for f such that $f(\tilde{t}_1,\ldots,\tilde{t}_n)$ and $f(t_1,\ldots,t_n)$ are unifiable.

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

Contextual narrowing is defined by the rule:

> If $M_i \leadsto_\sigma N_i$ for some $i \in \{1,\ldots,n\}$ and $\phi \in DC \cup FS$, then
>
> $$\phi(M_1,\ldots,M_i,\ldots,M_n) \leadsto_\sigma \phi(M_1\sigma,\ldots,N_i,\ldots,M_n\sigma)$$

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

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

2.4 Narrowing Strategies

The narrowing relation has two sources of nondeterminism. On the one hand, an expression may contain several narrowable subexpressions (***redex nondeterminism***). On the other hand, a function application containing free (logical) variables may be narrowable using different program rules (***rule nondeterminism***).

In an implementation, rule nondeterminism is usually handled by backtracking, i.e. the first applicable rule is chosen and the alternative rules and the state of the computation are retained in a so-called choice point. Whenever a failure occurs, the last choice point is inspected in order to reset the computation and to try the next alternative rule. Backtracking describes a a depth first search of the computation tree, which is of course incomplete, because it may fail to determine all possible narrowing results in the presence of infinite computations paths. Nevertheless, this technique is chosen in most implementations due to efficiency reasons.

The redex nondeterminism is usually handled by imposing a redex selection strategy. One distinguishes between innermost (eager) and outermost (lazy) strategies.

An innermost strategy restricts the application of the narrowing rule to innermost redexes (reducible expressions), i.e. to applications of a function symbol to terms, only. It is complete for confluent and terminating rewrite systems when all function symbols are totally defined [7]. As Mini-BABEL programs in general enable infinite derivations and partially defined functions, an innermost strategy is not adequate. Nevertheless, we will consider this strategy to the end of comparison.

In principle, an outermost or lazy strategy allows only the evaluation of outermost function applications. In order to guarantee completeness, it is however necessary to admit the evaluation of certain inner expressions. Sometimes it cannot be decided, whether a program rule can be applied to an outermost function application, because the unification of the rule's left hand side and the function application is successful, except for function applications in the arguments of the application meeting constructor terms in the left hand side of the rule. Such rules are called ***pending rules*** in [13]. They demand the evaluation of inner expressions of a function application.

Example: The rule "$f(0, s(Y)) := \ldots$" is pending for the application $f(0, g(X))$. Applicability of the rule can only be decided after the evaluation of the inner function application $g(X)$. $\triangleleft$

For this reason, a lazy strategy allows the evaluation of inner expressions, which are demanded by pending program rules. Contextual narrowing must be restricted to these cases. In [13] soundness and completeness results have been proved, which relate the narrowing semantics with a lazy selection strategy to a declarative semantics for first order BABEL.

3 Implementation of Functional Logic Languages

In this section we shortly describe the structure and the behaviour of the stack narrowing machine that has been introduced in [11]. The narrowing machine has been designed as the extension of a reduction machine for purely functional languages by mechanisms for performing unification and backtracking. The backtracking mechanism has been borrowed from Warren's Prolog Engine [18].

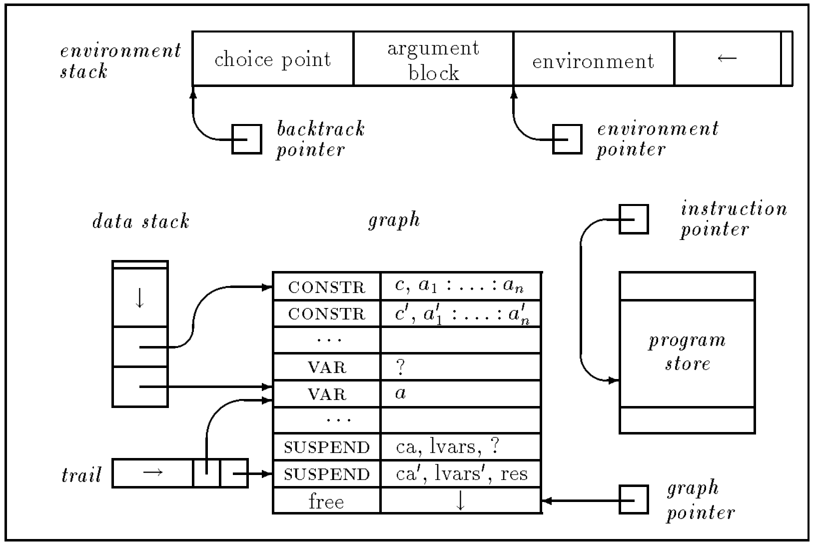

Figure 1: Structure of the Narrowing Machine

3.1 Structure of the Narrowing Machine

The store of the narrowing machine is illustrated in Fig. 1. Its central component is the environment stack which contains environments and argument blocks for the control of forward computations and choice points for the control of backward computations, i.e. backtracking.

Terms, i.e. structured terms and variables, and arguments, whose evaluation is postponed in case of a lazy evaluation strategy, are represented in the graph. Consequently, we distinguish between ***constructor nodes***, unbound and bound ***variable nodes***, and ***suspension nodes*** (see Fig. 1).

Suspension nodes need some further explanation. The code address ca is the address, at which the code for the evaluation of the delayed or suspended argument starts, ***lvars*** are the bindings of local variables, to which some instructions in the code may refer. The third component ***res*** is only necessary for the organization of backtracking. It equals '?' as long as the suspension node has not been evaluated. Due to the possibility of backtracking, the suspension node must not be overwritten by the result after its evaluation, as it is practice in implementations of functional languages. Instead the result address is noted in the suspension node, which behaves then just as an indirection node. When the computation fails and the state of the machine is reset to a state, where the suspension node had not been evaluated yet, the result component of the suspension node is reset to '?'.

The organization of delayed computations via suspension nodes is adopted from functional languages. Suspension nodes are a generalization of ***closures***, which have been introduced in [10] for the representation of functions. The closure concept has been adapted for the representation of arbitrary unevaluated expressions in [6]. The extension of the concept for functional logic languages has been introduced in [4].

The graph is organized as a global stack. It grows during forward computations and is reset on backtracking to the value of the graph pointer, which has been saved in the choice point.

The trail is used to keep note of bound variable nodes and updated suspension nodes in order to enable their reconstruction on backtracking. Resetting the trail means unbinding the variables and resetting the result component of the suspension nodes whose addresses have been noted in the trail.

3.2 Behaviour of the Narrowing Machine

In order to evaluate a function application, the addresses of the graph representations of the arguments are written on top of the data stack and a function call is executed. An ***environment*** and an ***argument block*** are allocated on top of the environment stack. The environment block contains locations for the local variables of the function rules, the saved pointer to the previously active environment, 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. The arguments on the data stack are moved into the argument block.

If the function symbol is defined by more than one rule, the defining rules are tested in their textual ordering. The machine code that is generated for such a function symbol is given in Fig. 2. First, the instruction 'TRY_ME_ELSE $rule_2$' creates a ***choice point*** on top of the argument block. The choice point contains the information that is necessary for restoring the current state of the machine on backtracking, i.e. among other things the current depth of the data stack, the length of the trail, the pointer to the previous choice point and especially the backtrack address, which indicates the code address of the next alternative rule, $rule_2$. Note that backtracking will either be initiated by a failing unification or a user's request for an alternative solution, after a solution has been determined.

After the creation of the choice point, the first function rule is applied to evaluate the application. The translation of a function 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 a successful unification (see Fig. 2). The unification of an argument and the formal parameter term of the function rule is done by traversing the parameter term top down left to right and performing local comparisons using the instructions MATCHVAR and MATCHCONSTR (see compilation scheme ***matchtrans*** in Fig. 3). The evaluation of arguments, which are represented as suspension nodes, is started by the instruction INITIATE, whenever these must be unified with constructor terms. If no failure occurs during the unification phase, the POP_ARGS-instruction removes the argument block from the stack, unless a choice point is on top of the stack. Then, the rule's right hand side is evaluated. Finally, the RETURN instruction successfully terminates the function call, by giving the control back to the previously active environment and by setting the instruction pointer to the return address stored in the environment.

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

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathit{proctrans}(\langle f(t_{i1},\dots,t_{in}) := M_i.} \\
\multicolumn{2}{r}{\mid 1 \le i \le r, r \ge 2\rangle)} \\
:= & \text{TRY_ME_ELSE } \mathit{rule}_2 \\
 & \mathit{ruletrans}\ (f(t_{11},\dots,t_{1n}) := M_1.) \\
\mathit{rule}_2: & \text{RETRY_ME_ELSE } \mathit{rule}_3 \\
 & \mathit{ruletrans}\ (f(t_{21},\dots,t_{2n}) := M_2.) \\
\mathit{rule}_3: & \vdots \\
 & \vdots \\
\mathit{rule}_r: & \text{TRUST_ME_ELSE_FAIL} \\
 & \mathit{ruletrans}\ (f(t_{r1},\dots,t_{rn}) := M_r.)
\end{array}
\qquad
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{ruletrans}(f(t_1,\dots,t_n) := M.)} \\
:= & \text{LOAD } 1 \\
 & \mathit{matchtrans}(t_1) \\
 & \vdots \\
 & \text{LOAD } n \\
 & \mathit{matchtrans}(t_n) \\
 & \text{POP_ARGS} \\
 & \mathit{exptrans}\ (M) \\
 & \text{RETURN}
\end{array}$$

Figure 2: Compilation of Procedures and Rules

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathit{matchtrans}(X_i)} \\
:= & \text{MATCHVAR } i
\end{array}
\qquad
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{matchtrans}(c(t_1,\dots,t_n))} \\
:= & \text{INITIATE} \\
 & \text{MATCHCONSTR } (c,n) \\
 & \mathit{matchtrans}(t_1) \\
 & \vdots \\
 & \mathit{matchtrans}(t_n)
\end{array}$$

Figure 3: Code for Unification

If the unification of the parameter terms and arguments is not successful, backtracking is initiated which uses the information given in the top level choice point to reset the components of the machine and to jump to the backtrack address. If there are still more alternative rules for the function, the RETRY_ME_ELSE $rule_i$ instruction updates the backtrack address within the choice point. If the next alternative rule is the last defining rule for the function symbol, the choice point is removed from the environment stack by the TRUST_ME_ELSE_FAIL instruction (see Fig. 2). The latter has the effect that backtracking will subsequently reset the computation to the topmost choice point below the current environment, which belongs to a function application whose evaluation terminated successfully, but for which there are still alternative function rules which have not been tried yet. If there is no more choice point on the environment stack, a failure will cause the overall computation to fail.

The narrowing machine is capable of realizing an innermost and a lazy strategy. Which strategy is chosen to execute a program, depends on the compilation of argument expressions. In case of an innermost strategy, arguments are translated by the ***exptrans***-scheme. This implies that expressions are evaluated to normal form and INITIATE instructions can be omitted. In case of a lazy evaluation strategy, arguments are translated by the ***delaytrans***-scheme, which generates suspension nodes for the representation of function calls using the instruction SUSPEND. Furthermore, code for

$\mathit{exptrans}(X_i)$	$\mathit{delaytrans}(X_i)$
$:=$ LOAD i	$:=$ LOAD i
INITIATE	$\mathit{delaytrans}(c(M_1,\ldots,M_n))$
$\mathit{exptrans}(c(M_1,\ldots,M_n))$	$:=$ $\mathit{delaytrans}(M_1)$
$:=$ $\mathit{argtrans}(M_1)$	$\vdots$
$\vdots$	$\mathit{delaytrans}(M_n)$
$\mathit{argtrans}(M_n)$	NODE (c,n)
NODE (c,n)	$\mathit{delaytrans}(f(M_1,\ldots,M_n))$
$\mathit{exptrans}(f(M_1,\ldots,M_n))$	$:=$ SUSPEND l
$:=$ $\mathit{argtrans}(M_1)$	JUMP l'
$\vdots$	l: $\mathit{exptrans}(f(M_1,\ldots,M_n))$
$\mathit{argtrans}(M_n)$	UPDATE
CALL (f,n)	l': $\ldots$

$$\mathit{argtrans}(M) := \begin{cases} \mathit{exptrans}(M) & \text{in case of an “eager” evaluation strategy} \\ \mathit{delaytrans}(M) & \text{in case of a “lazy” evaluation strategy} \end{cases}$$

Figure 4: Code for Unification and Compilation of Expressions

the evaluation of the suspension is generated using the *exptrans*-scheme. This code is concluded by the instruction UPDATE, which writes the result of the evaluation into the suspension node and gives control back to the calling environment. A lazy evaluation strategy evaluates expressions to head normal form, i.e. until the top level constructor of the result is determined. In order to force the complete evaluation of the goal expression, a special mechanism must be incorporated into the machine.

This gives an impression of the principal behaviour of the narrowing machine. For simplicity, we have not considered any optimization techniques as e.g. tail recursive function calls and the dynamic detection of determinism. The interested reader is referred to [11] and [12] for further details.

4 Disadvantages of Lazy Narrowing

The lazy narrowing implementation described in the last section has several disadvantages which are due to the interaction of lazy evaluation and backtracking. As argument evaluations are initiated during unification, the environments and choice points of the arguments are allocated on top of the environment and choice point of the function call (see Fig. 5). Thus, a failing unification causes backtracking on the argument evaluations and all alternative evaluations of arguments are tried for a fixed function rule, before the next alternative function rule is selected. To put it clearly, arguments may be completely re-evaluated for each function rule.

Example: Consider the following definition of the function symbols *even* and *odd*:

$$\begin{array}{lll} even(0) & := & s(0). \\ even(s(s(X))) & := & even(X). \qquad odd(X) \;:=\; even(even(X)). \\ even(s(0)) & := & 0. \end{array}$$

Lazy Narrowing: "‘Try all alternative argument evaluations for a fixed function rule."’

$$\underbrace{\boxed{cp(M_{i_l})}\boxed{env(M_{i_l})}\boxed{\cdots\cdots}\boxed{cp(M_{i_1})}\boxed{env(M_{i_1})}}_{\text{Demanded Arguments}}\underbrace{\boxed{cp(f)}\boxed{env(f)}}_{\text{Function Call}}\boxed{\cdots\leftarrow\cdots}$$

Innermost Narrowing: "‘Try all function rules for a fixed argument evaluation."’

$$\underbrace{\boxed{cp(f)}\boxed{env(f)}}_{\text{Function Call}}\underbrace{\boxed{cp(M_n)}\boxed{env(M_n)}\boxed{\cdots\cdots}\boxed{cp(M_1)}\boxed{env(M_1)}}_{\text{Arguments}}\boxed{\cdots\leftarrow\cdots}$$

Figure 5: Structure of the Environment Stack for the Evaluation of $f(M_1,\ldots,M_n)$

During the lazy execution of the expression $odd(s(s(0)))$ the argument expression $even(s(s(0)))$ will be evaluated three times, once for each defining rule of *even*. ◁

Multiple argument evaluations are caused by the resetting of suspension nodes on backtracking. But this resetting is only necessary, if the evaluation of the suspension node depends on logical variables, which are also reset. Suspension nodes can only depend on logical variables, which existed already when the node was generated, i.e. the graph address of which is less than the graph address of the suspension node. This immediately leads to the following

Refinement of the Backtracking Mechanism:
Reset suspension nodes on backtracking, only if a variable, the address of which is less than the address of the suspension node and which has been noted in the trail before the suspension node, is reset.

This refinement applies only to suspension nodes, the evaluation of which is locally deterministic. If a suspension node is not reset, the graph cannot be reset on backtracking, too, because the result of the suspension node must be kept.

Thus, the problem of superfluous argument re-evaluations can be avoided by a careful treatment of suspension nodes on backtracking. A more severe problem is however the divergence of lazy function calls, if an argument evaluation yields infinitely many alternative solutions which do not match the pattern of the selected function rule.

Example: The program rules

$$\begin{array}{lll} one(0) & := & s(0). \\ one(s(N)) & := & one(N). \end{array}$$

define the unary function *one*, that maps all constructor terms built from the nullary constructor 0 and the unary constructor s to the constructor term $s(0)$.

Note that there are infinitely many narrowing derivations starting from the expression $one(X)$, where X is a logical variable:

$$\forall\, i \in \mathbb{N}:\ one(X) \rightsquigarrow^{*}_{\{X/\underbrace{s(\ldots s(}_{i\ \text{times}}0)\ldots)\}} s(0).$$

Each derivation yields the result $s(0)$, while the variable X is bound to a term of the form $s(\ldots s(0)\ldots)$. An innermost evaluation of the expression $one(one(X))$ has the same outcomes.

A lazy evaluation of the same expression with backtracking will however fail to terminate. This is due to the fact, that the system tries to apply the first rule of the *one*-function with the argument not evaluated yet. The evaluation of the argument yields the result $s(0)$ binding the logical variable X to 0. The unification of 0 and $s(0)$ fails and backtracking is initiated that leads to the re-evaluation of the argument. As there are infinitely many alternative evaluations of the argument yielding the same result, this computation will not terminate. ◁

Note that the nontermination problems are not restricted to the case of circumstantial definitions of constant functions.

Example: Consider the following definitions of the arithmetic operations for addition and multiplication:

$$\begin{array}{lcllcl} add(0,Y) & := & Y. & mult(0,Y) & := & 0. \\ add(s(X),Y) & := & add(X,s(Y)). & mult(s(X),Y) & := & add(Y,mult(X,Y)). \end{array}$$

With these definitions, the lazy narrowing evaluation of the expression

$$mult(add(X,s(0)),s(s(0)))$$

will not terminate.

Replacing the second rule for *add* by "$add(s(X),Y) := s(add(X,Y))$." would avoid the nontermination, but in general the tail recursive version is preferred as it is more efficient. ◁

Simply changing the order of backtracking is in general not possible, because the evaluation of arguments may depend on logical variables which have been bound to pattern of the selected function rule. The only way to avoid the nontermination problem, is to evaluate arguments before function calls (i.e. using an innermost evaluation strategy). The innermost evaluation strategy should of course only be used for demanded arguments of function calls, i.e. for arguments that are necessary to determine the result of the application. In the following section, we will define a notion of demandedness, which implies strictness of functions. Strictness is a semantic property of functions. Let $\mathcal{A}_i = \langle A_i, \leq_i\rangle$ be complete partial orders with least elements $\perp_i$ ($0 \leq i \leq n, n \in \mathbb{N}$). A function $f : A_1 \times \ldots \times A_n \to A_0$ is called ***strict in its ith argument***, if for all $a_j \in A_j$ ($1 \leq j \leq n$):

$$f(a_1,\ldots,a_{i-1},\perp_i,a_{i+1},\ldots,a_n) = \perp_0,$$

i.e. whenever the ith argument is not defined, the application of f is not defined.

In the context of purely functional languages, it is safe to evaluate strict arguments before evaluating the function application. Thus, the use of an innermost evaluation strategy for the strict arguments of functions is a well-known optimization technique for the implementation of purely functional languages [15]. A straightforward adaptation of the strictness analysis methods, which have been developed for purely functional languages, to BABEL is however only possible, if a pattern matching compiler is used to transform the programs. But such a compiler does not always preserve the semantics of programs. Therefore, we will develop an analysis method, that directly works on function symbols defined by several rules.

5 Incorporation of Demandedness Information

For simplicity, we will assume that Mini-BABEL programs are transformed into equivalent programs with flat left hand side, i.e. the term pattern of the left hand side of rules are variables or constructors applied to variables. Such a transformation is always possible by the introduction of auxiliary function symbols [14].

For a given program with flat left hand sides, we will now try to find out, which arguments are demanded by the functions defined in the program. For each $f \in FS$ we will compute a set $\mathcal{D}(f) \subseteq \mathbb{N}$, which will contain the argument positions, which are needed for the evaluation of an application of f.

Let $f \in FS$. The demandedness of f's arguments depends on the structure of the defining rules for f. A single rule demands the evaluation of an argument, if the corresponding term pattern of the rule's left hand side (lhs) is non-variable or if the evaluation of the argument is demanded by the rule's right hand side (rhs). The latter is the case, when the argument variable occurs in a demanded argument position of another function symbol or if the right hand side is identical to the argument variable. Whenever the right hand side of a rule is a constructor application, no arguments are demanded by it. We will say that f demands an argument when this argument is demanded by all defining rules of f. Formally, the sets of demanded arguments of the function symbols defined in a program are specified by the following recursive equation system.

Let $\mathcal{P} = \langle\langle f^j(t^j_{i1},\ldots,t^j_{in_j}) := M^j_i \mid 1 \leq i \leq r_j\rangle \mid 1 \leq j \leq m\rangle$ be a Mini-BABEL program which defines the function symbols f^j. The sets $\mathcal{D}(f^j) \subseteq \mathbb{N}$ of ***demanded arguments of f^j*** are determined by the least fixpoint of the following system of recursive set equations:

$$\langle \mathcal{D}(f^j) = \bigcap_{i=1}^{r_j} \overbrace{(\underbrace{\{k \mid t^j_{ik} \notin Var\}}_{\text{demanded by lhs}} \cup \underbrace{\{k \mid t^j_{ik} \in Var \wedge t^j_{ik} \in DVar(M^j_i)\}}_{\text{demanded by rhs}})}^{\text{demanded arguments of rule } i} \mid 1 \leq j \leq m\rangle$$

where

$$\begin{array}{lcll} DVar(X) & := & \{X\} & \%\ X \in Var \\ DVar(c(M_1,\ldots,M_n)) & := & \emptyset & \%\ c \in DC \\ DVar(f(M_1,\ldots,M_n)) & := & \bigcup_{l \in \mathcal{D}(f)} DVar(M_l) & \%\ f \in FS. \end{array}$$

Note that this least fixpoint is well-defined in the complete partial order $\langle\mathcal{P}(\mathbb{N}),\subseteq\rangle$, where $\mathcal{P}(\mathbb{N})$ denotes the powerset of $\mathbb{N}$. It can be computed following the well-known theorem of Knaster-Tarski by iteratively computing approximations of the sets $\mathcal{D}(f^j)$ ($1 \leq j \leq m$), starting with $\emptyset$ for each set, until a stagnation occurs.

Example: For the example programs of the previous section, we get the following demandedness information: $\mathcal{D}(one) = \mathcal{D}(add) = \mathcal{D}(mult) = \{1\}$. ◁

Demandedness implies strictness with respect to the declarative semantics of BABEL defined in [13], i.e. demandedness is a computable approximation of strictness information and the demandedness analysis corresponds to a strictness analysis.

The demandedness information makes it possible to switch from a purely lazy evaluation strategy to a mixture of the eager and the lazy strategy by using the innermost evaluation strategy for demanded arguments and the lazy strategy for non-demanded arguments, as it is usually done in the context of functional languages. This means for the compilation of Mini-BABEL that we modify the *argtrans*-scheme in the following way:

$$argtrans(M) := \begin{cases} exptrans(M) & \text{if } M \text{ is a demanded argument} \\ & \text{of a function application} \\ delaytrans(M) & \text{otherwise} \end{cases}$$

The mixed strategy has several advantages with respect to the purely lazy strategy. It is more efficient, because the (time and space) overhead of producing suspension nodes for the representation of demanded arguments is saved. This is confirmed by the implementation results given in the next section. Its advantage with respect to an innermost strategy is that it does not evaluate unnecessary subexpressions. Furthermore, it solves the nontermination problems for a large class of programs, in particular for the examples of the previous section.

The following example shows that it is however not possible to define a ***safe*** evaluation strategy for arbitrary programs, that avoids the nontermination problems caused by the interaction of lazy evaluation and backtracking.

Example: Let $DC^{(\varepsilon,bool)} := \{tt, ff\}$. Consider the following program rules:

$$\begin{array}{lllllll} cond(tt,X) & := & s(X). & & one\_zero(s(X)) & := & zero\_zero(X). \\ cond(ff,X) & := & s(0). & & zero\_zero(0) & := & 0. \end{array}$$

The demandedness analysis yields $\mathcal{D}(cond) = \mathcal{D}(one\_zero) = \mathcal{D}(zero\_zero) = \{1\}$.

Thus the mixed strategy evaluates all argument expressions eagerly, except for the second argument of *cond*. But the only possibility to avoid the nontermination caused by the delayed evaluation of the first argument of *cond* in the expression $one\_zero(cond(B, one(X)))$ is to evaluate this argument before evaluating the application of *cond*. Only the innermost evaluation strategy yields the result 0 binding X to 0 and B to ff. ◁

This example reveals that strictness information is not sufficient to solve the nontermination problems in general. The following lemma contains a syntactic characterization of a program class, for which the mixed strategy is safe and avoids the nontermination problems.

Lemma: Let $\mathcal{P} = \langle\langle f^j(t^j_{i1}, \ldots, t^j_{in_j}) := M^j_i \mid 1 \le i \le r_j\rangle \mid 1 \le j \le m\rangle$ be a Mini-BABEL program.

If for each $j \in \{1, \ldots, m\}$, f^j is totally defined and for each non-demanded argument position of f^j, $k \in \{1, \ldots, n_j\} \setminus \mathcal{D}(f^j)$: $\{t^j_{1k}, \ldots, t^j_{r_jk}\} \subseteq Var$, then the mixed strategy guarantees that divergence cannot occur during rule selection. ◁

Example Program	Runtimes without	with
	refined backtracking	
$quicksort([5,..,1])$	0.42 sec	0.089 sec
$quicksort([6,..,1])$	2.33 sec	0.121 sec
$quicksort([7,..,1])$	13.41 sec	0.158 sec
$quicksort([8,..,1])$	79.37 sec	0.201 sec

Table 1: Saving of Runtimes by Refined Backtracking

Proof. As function symbols are totally defined, there will be always a rule that is applicable. Additionally, no argument evaluation is ever initiated during the unification phase, as demanded arguments are evaluated before function calls and non-demanded arguments always meet variable parameters. □

The consistency of our demandedness notion and the semantic strictness notion is due to the fact that the standard semantics of BABEL does not distinguish between non-definedness caused by nontermination and non-definedness caused by partially defined functions, usually called finite failure. For deterministic computations and in purely functional languages the identification of the two non-definedness situations makes no problems. Occurrence of either means the non-definedness of the result of the overall computation. In the context of functional logic languages, the replacement of finite failure and nontermination may however lead to the incompleteness of the evaluation mechanism. A finite failure may cause the inspection of alternative computation paths, while nontermination leads to the nontermination of the overall computation, at least in a sequential system.

The evaluation of demanded arguments of functions is only safe, if function symbols are totally defined. For partially specified functions the demandedness notion must be refined in such a way that it corresponds to strictness with respect to a denotational semantics which distinguishes finite failure from divergence. The discussion of this refinement is however out of the scope of this paper.

6 Implementation Results

The runtimes given in this section have been measured using an implementation of full BABEL, which consists of a compiler frontend that translates BABEL programs into code for the abstract narrowing machine, and an interpreter for the abstract machine code, written in C. The implementation supports natural numbers and the usual arithmetic operations as primitives. All examples have been executed on a SPARC station 1+ (Sun-4/65).

Table 1 shows the runtimes that are saved by the refined backtracking mechanism for the functional quicksort algorithm. As this algorithm is deterministic, updated suspension nodes need not be reset on backtracking and thus useless recomputations of argument expressions can be avoided.

Example Program	Evaluation Strategies innermost	mixed	lazy
fib_nat 15	0.191 sec	0.191 sec	0.254 sec
fib_nat 20	2.048 sec	2.048 sec	2.818 sec
fib_list 15	∞	5.00 msec	5.68 msec
fib_list 20	∞	6.40 msec	7.30 msec
queens 8	9.244 sec	9.906 sec	10.342 sec
solitaire	826 sec	847 sec	856 sec

Table 2: Comparison of Strategies

Currently, the compiler allows the programmer to specify for each function the demanded arguments. Thus, it is possible to compare the various evaluation strategies.

The execution times of the innermost, the lazy and the mixed evaluation strategy are compared in Table 2 for some example programs: *fib_nat* denotes the recursive definition of the fibonacci function for natural numbers; *fib_list* selects elements from the infinite list of fibonacci numbers; *queens n* is the problem of finding all possibilities to place n queens on an $n \times n$ chessboard in such a way that they cannot beat each other, and *solitaire* is a simple game program.

On the one hand, the runtimes show the overhead of delayed evaluations, when the suspensions are almost always evaluated. The mixed strategy saves time by the eager evaluation of demanded arguments. On the other hand, lazy evaluation admits a different style of programming using infinite data structures, as indicated by the *fib_list*-example.

Note that the order, in which solutions are produced for nondeterministic programs, varies from strategy to strategy, because the backtracking behaviour is altered by the different argument evaluation strategies, as discussed in the previous sections.

7 Related Work and Conclusions

The lazy narrowing implementation described in [3] does not take notice of the nontermination problems caused by the interaction of lazy evaluation and backtracking. For the first time, these problems have been noted and discussed in [14]. In order to avoid them, a program transformation to so-called uniform programs has been proposed. The demandedness analysis presented in Sect. 5 can be seen as a continuation and improvement of the approach of uniform programs. The purely syntactic notion of demanded arguments introduced in [14] has been extended to a more semantics based notion, which has the advantages that demandedness is invariant with respect to equivalent program transformations, and that the special nontermination problems of lazy narrowing are avoided more often.

In [4] the pattern matching compiler described in [15] is used to translate programs into an intermediate form, which allows a deterministic selection of rules for non-variable arguments. This method also sometimes avoids the nontermination problems. But the example, for which the nontermination can only be avoided by the innermost evaluation of a non-demanded argument, will also not terminate.

To sum up, the interaction of lazy evaluation and backtracking has several disadvantageous effects, which can be avoided by postponing the resetting of suspension nodes during backtracking as long as possible and by incorporating demandedness information of function arguments into the evaluation strategy. Unfortunately, the nontermination caused by the backtracking strategy and the nontermination caused by the evaluation of non-demanded arguments cannot both be avoided for arbitrary programs by a uniform strategy. Nevertheless, we believe that the strategy based on the demandedness information is a reasonable compromise between an innermost and a lazy strategy that will meet the requirements of most real-life programs.

Acknowledgements

We are grateful to Michael Hanus and Andreas Mück for helpful comments on a previous version of this paper.

References

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

[2] P. G. Bosco, E. Giovannetti: *IDEAL: An Ideal Deductive Applicative Language*, IEEE Symp. on Logic Progr. 1986, IEEE Comp. Soc. Press, 89-94.

[3] 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. 1989, MIT Press 1989, 318–333.

[4] M. M. T. Chakravarty, H. C. R. Lock: *The Implementation of Lazy Implementation*, Symp. on Progr. Language Impl. and Logic Progr. (PLILP) 1991, LNCS 528, Springer Verlag 1991, 123–134.

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

[6] J. Fairbairn, St. Wray: *TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators*, Conf. on Func. Progr. Languages and Computer Arch. 1987, LNCS 274, Springer Verlag 1987, 34–45.

[7] L. Fribourg: *SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting*, IEEE Symp. on Logic Progr. 1985, IEEE Computer Society Press 1985, 172–184.

[8] M. Hanus: *Compiling Logic Programs with Equality*, Workshop on Progr. Language Impl. and Logic Progr. (PLILP) 1990, LNCS 456, Springer Verlag 1990, 387–401.

[9] P. Hudak, P. Wadler: *Report on the Functional Programming Language Haskell*, SIGPLAN Notices, Vol. 27, No. 5, May 1992.

[10] P. J. Landin: *The Mechanical Evaluation of Expressions*, Computer Journal, 6, 4, 1964.

[11] R. Loogen: *From Reduction Machines to Narrowing Machines*, TAPSOFT 1991, LNCS 494, Springer Verlag 1991, 438–457.

[12] R. Loogen, St. Winkler: *Dynamic Detection of Determinism in Functional Logic Languages*, Symp. on Progr. Language Impl. and Logic Progr. (PLILP) 1991, LNCS 528, Springer Verlag 1991, 335–346.

[13] J. J. Moreno-Navarro, M. Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Progr., Vol. 12, North Holland 1992, 191–223.

[14] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: *Lazy Narrowing in a Graph Machine*, Conf. on Algebraic and Logic Progr. 1990, LNCS 463, Springer Verlag 1990, 298–317.

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

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

[17] D. Turner: *Miranda: A non-strict functional language with polymorphic types*, Conf. on Func. Progr. Lang. and Computer Arch. 1985, LNCS 201, Springer Verlag 1985, 1–16.

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