Full Indexing for Lazy Narrowing

Werner Hans Herbert Kuchen Stephan Winkler
RWTH Aachen, Lehrstuhl für Informatik II*

Abstract

Indexing improves the efficiency of many logic programs considerably. Here, we present a new indexing scheme for a functional logic language which has lazy narrowing as operational semantics. The main difficulty is that the arguments of a function may be unevaluated and that the indexer has to incrementally initiate the evaluation of the arguments to head normal form, if this is required for the selection of a rule. Moreover, each argument has to be evaluated as early as possible in order to avoid re-evaluations of arguments due to backtracking.

Keywords: Indexing, Narrowing, Functional Logic Programming, Logical Extensions.

1 Introduction

Indexing is an important technique for improving the efficiency of logic programs. The idea is to avoid the unification of the considered goal with the left hand sides of those rules, where it is clear that this unification is going to fail, and to access applicable rules directly by using information about the arguments. If only one rule remains applicable, the computation is (locally) deterministic and a choice point is not needed at this place. If no rule remains applicable, backtracking can be initiated directly.

Several indexing schemes have been proposed which differ in the number of arguments they consider (e.g. one [AK91] or all) and the auxiliary structures used, e.g. decision trees, decision graphs. Decision trees can cause an exponential code explosion. By using decision graphs [KS88, KT91, HM89] common parts can be shared, reducing the code explosion. Another alternative [PN91, Ha92] is to handle all the arguments separately and to compute the intersection of the sets of rules, which remain applicable after investigating each individual argument. All these approaches rely on the fact that the arguments of a predicate are known when the indexing scheme is applied.

An extension of logic programming languages are so-called functional logic languages [BL86, DL86]. They have a syntax like purely functional languages but use narrowing – a unification based parameter passing mechanism which subsumes rewriting and SLD-resolution – as operational semantics. Several narrowing strategies can be considered. Eager narrowing evaluates the arguments of a function before the function is applied. For eager narrowing the indexing schemes for logic languages are (almost) directly applicable. More interesting is a lazy narrowing strategy, which allows to handle infinite data structures and

*D-52056 Aachen, Germany, {hans,herbert,winkler}@zeus.informatik.rwth-aachen.de; this work was supported by DFG-grant In 20/6-1.

a more modular style of programming (see [Hu84] for an overview of the advantages of lazy evaluation). Lazy narrowing evaluates the arguments of a function as late as possible and only when they are demanded. An argument is *demanded* if it has to be unified with a constructor term t on the left hand side of a rule. In this case, it is evaluated to so-called *head normal form* (hnf), i.e. to an expression which is an unbound variable or which has a constructor at the top (and some possibly unevaluated subexpressions). If the topmost constructor matches, the subexpressions are unified analogously with the corresponding subterms of t.

For lazy narrowing the existing indexing schemes cannot be used, since they expect the arguments to be evaluated. Instead, an indexing scheme is needed which incrementally initiates the evaluation of (parts of) the arguments to hnf, if this is needed in order to decide which rules are applicable. In this paper, we will present such an indexing scheme. It uses a decision graph (although for the simplicity of presentation we will use decision trees here) as auxiliary structure. Moreover, the indexing code has to initiate the evaluation of the arguments of a function as early as safely possible in order to avoid re-evaluations of arguments. This will be explained in detail in Section 3.

This paper is organized as follows. In Section 2, we present the functional logic language Babel which serves as an example language with lazy narrowing as operational semantics. In Section 3, we discuss the basic ideas and some pitfalls for the implementation of lazy narrowing. Section 4 contains the formal specification of the new indexing scheme and a corresponding example. Section 5 shows some experimental results. In Section 6, we conclude and point out future work.

2 Lazy Babel

The language Babel integrates functional and logic programming in a flexible and mathematically well-founded way. It is based on a constructor discipline and uses narrowing as evaluation mechanism [Re85]. Babel has a polymorphic type system, similar to that of [Mi78]. For simplicity of presentation, we will here ignore higher order functions and polymorphic types, since they are not essential for the present paper.

2.1 The Syntax of Babel

Rather than giving a precise description of the syntax and semantics of Babel (which can be found in [MR92, KLMR92]) we will first show a small example program and then present some basic concepts. In the example, a function *deriv* is defined, which computes the derivation of a function $f(x) = e_1$. More precisely: *deriv* gets the body e_1 of f as an argument and delivers an expression e_2 such that $f'(x) = e_2$.

$$
\begin{array}{l}
\textbf{datatype}\ expr := x \mid const\ int \mid add\ expr\ expr \mid mul\ expr\ expr \mid exp\ expr \mid \ldots. \\
\textbf{fun}\ deriv\text{: } expr \to expr. \\
\quad \begin{array}{ll}
deriv\ x & := const\ 1. \\
deriv\ (const\ N) & := const\ 0. \\
deriv\ (add\ U\ V) & := simplify\ (add\ (deriv\ U)\ (deriv\ V)). \\
deriv\ (mul\ U\ V) & := simplify\ (add\ (mul\ U\ (deriv\ V))\ (mul\ V\ (deriv\ U)). \\
deriv\ (exp\ U) & := simplify\ (mul\ (exp\ U)\ (deriv\ U)). \\
\ldots \text{other rules for } deriv \ldots &
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun}\ simplify\text{: } expr \to expr. \\
\quad \begin{array}{ll}
simplify\ x & := x. \\
simplify\ (mul\ E\ (const\ 1)) & := simplify\ E. \\
simplify\ (exp\ E) & := exp\ (simplify\ E). \\
\ldots \text{other rules for } simplify \ldots &
\end{array}
\end{array}
$$

A Babel *program* consists of a sequence of datatype definitions and a sequence of function definitions. The program can be queried with a goal expression. For example, a valid query for the previous program is

$$\textbf{solve}\ deriv\ E = E \to true.$$

This goal corresponds to the differential equation $f'(x) = f(x)$. A possible result for this goal is *true* provided that the variable E is bound to $(const\ 0)$. Another solution consists of the result *true* and the substitution of E by $exp\ x$.

A **datatype** definition introduces a new *type*, e.g. *expr*, and a sequence of *data constructors* (separated by |; e.g. x (of arity 0) and *const* (of arity 1)). A *data constructor* is an uninterpreted function mapping its argument types (given behind the name of the constructor) to the type on the left hand side of the datatype definition. For example, *const* has the type $const\text{: } int \to expr$. A Prolog-like syntax for lists is allowed, i.e. $[e \mid e']$ represents the list with head e and tail e', and [ ] is the empty list.

For each function, its type and a sequence of defining rules are specified. A *rule* for a function f has the form

$$\underbrace{f\ t_1 \ldots t_m}_{\text{left hand side}} \ := \ \underbrace{\underbrace{\{b \to\}}_{\text{optional guard}} \quad \underbrace{e}_{\text{body}}}_{\text{right hand side}} \ .$$

where $t_1,\ldots,t_n$ are terms sharing no variables (left linear), b is a boolean expression, and e is an arbitrary expression. A *term* is either a (logical) *variable* (denoted by an identifier beginning with a capital letter) or an application of a n-ary data constructor ($n \geq 0$) to n argument terms:

$$
\begin{array}{ll}
t ::= X & \%\ \text{variable} \\
\quad \mid (c\ t_1 \ldots t_n) & \%\ \text{application of a } n\text{-ary data constructor } c\ (n \geq 0) \text{ to } n \text{ terms}
\end{array}
$$

An *expression* has the form:

$$
\begin{array}{ll}
e ::= X & \%\ \text{variable} \\
\quad \mid (\varphi\ e_1 \ldots e_n) & \%\ \text{application of a } n\text{-ary data constructor/function } \varphi\ (n \geq 0) \\
\quad \mid (b \to e) & \%\ \text{guarded expression, meaning ``}\textit{if } b \textit{ then } e \textit{ else undefined}\text{''} \\
\quad \mid (b \to e_1 \ \square\ e_2) & \%\ \text{conditional expression, meaning ``}\textit{if } b \textit{ then } e_1 \textit{ else } e_2\text{''}
\end{array}
$$

All expressions have to be well typed. Parentheses can be omitted, if the structure of an expression is clear.

Babel functions are functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is at most one result. This is ensured by syntactic restrictions for the rules [MR92]. Another restriction is left linearity allowing only unique variable occurrences on the left hand side of the rules. This simplifies the specification somewhat. The types *bool* (with data constructors *true* and *false*) and *int* (integer numbers) are predefined. Moreover, several basic (infix and mixfix) operations like equality (=), conjunction (**,**), disjunction (**;**), negation ($\neg$), and so on are predefined.

2.2 Narrowing Semantics

Here, we will present an operational semantics of Babel based on *narrowing*. To use narrowing as the operational semantics of functional logic languages was first proposed by Reddy [Re85].

Let $e \equiv f\ e_1 \ldots e_n$ be an expression and let $f\ t_1 \ldots t_n := e'$ be a variant of a rule sharing no variables with e. Moreover, let $\sigma \circ \lambda$ be the *most general unifier* of e and $(f\ t_1 \ldots t_n)$, i.e. the minimal substitution (of variables by expressions), such that $e\sigma$ is syntactically identical to $(f\ t_1 \ldots t_n)\lambda$. Then, e can be *narrowed* (in one step) to $e'' \equiv e'\lambda$ with *answer substitution* σ (denoted by $e \Rightarrow_\sigma e''$). Moreover, if for some i $(1 \leq i \leq n)$ $e_i \Rightarrow_\sigma e'_i$, then $(\varphi\ e_1 \ldots e_i \ldots e_n) \Rightarrow_\sigma (\varphi\ e_1\sigma \ldots e'_i \ldots e_1\sigma)$ where φ is a n-ary defined function or a n-ary constructor.

If, after several steps, an expression e is narrowed to a term t, we speak of a *computation* for the *goal* e with *result* t and *answer* σ, where σ is the composition of the answer substitutions of all the individual steps. If e cannot be narrowed further, but it is not a term, then the computation *fails*. The implementation will backtrack in such a situation. The rules are tried in their textual order.

In the present paper, we consider a lazy narrowing strategy [MKLR90, HLW92]. This strategy narrows the leftmost outermost redex, unless an inner redex needs to be narrowed first in order to decide whether the currently considered rule (following the textual order of rules) can be applied or not.

Alternatively, an innermost (eager) narrowing strategy [**?**] can be considered, where all arguments of a function have to be narrowed before a rule for the function can be applied.

The goal $\mathit{deriv}\ E = E \rightarrow \mathit{true}$ in the above example can be (lazily) narrowed as follows:

$$\begin{array}{lll} \mathit{deriv}\ E = E \rightarrow \mathit{true} & \Rightarrow_{\{E/x\}} & \text{(rule 1)} \\ \mathit{const}\ 1 = x \rightarrow \mathit{true} & \Rightarrow_{\{\}} & \text{(rule for equality)} \\ \mathit{false} \rightarrow \mathit{true} & & \end{array}$$

This computation fails, since there is no rule for $\rightarrow$ with first argument *false*. Another computation is:

$$\begin{array}{lll} \mathit{deriv}\ E = E \rightarrow \mathit{true} & \Rightarrow_{\{E/(\mathit{const}\ N)\}} & \text{(rule 2)} \\ \mathit{const}\ 0 = \mathit{const}\ N \rightarrow \mathit{true} & \Rightarrow_{\{\}} & \text{(rule for equality)} \\ 0 = N \rightarrow \mathit{true} & \Rightarrow_{\{N/0\}} & \text{(rule for equality)} \\ \mathit{true} \rightarrow \mathit{true} & \Rightarrow_{\{\}} & \text{(rule for } \rightarrow\text{)} \\ \mathit{true} & & \end{array}$$

Hence, this computation leads to a solution consisting of the result *true* and the answer $\{E/(\mathit{const}\ 0)\}$. By applying other rules for *deriv* to the goal, more solutions can be reached, e.g. the result *true* with answer $\{E/(\mathit{exp}\ x)\}$.

A precise definition of the operational semantics of Babel is found in [MR92]. This paper also contains the declarative semantics as well as correctness and completeness results.

3 Implementing Lazy Narrowing

Next, we would like to discuss the basic concepts for an efficient implementation of lazy narrowing. Several approaches (e.g. [MKLR90, HLW92, JMM92]) have been proposed which all claim to be lazy, but differ in the degree of laziness reached.

We choose as a starting point the naive lazy narrowing scheme, which evaluates a function application $f e_1 \dots e_n$ as follows:

- the rules for f are tried in their textual order
- for a given rule $f\ t_1 \dots t_n := e$, the arguments $e_1, \dots, e_n$ are unified from left to right with the corresponding terms $t_1, \dots, t_n$. If such an unification is not yet possible, since the considered expression e_i is not evaluated far enough, e_i is evaluated to some e_i' which is evaluated far enough. If the unification of e_i and t_i fails, alternative solutions for e_i are searched by backtracking. If no solution for e_i could be found, which is unifiable with t_i, the arguments $e_{i+1}, \dots, e_n$ are not evaluated at all, but the next rule is tried.

Let us explain this using an example: The first function *sprod* computes the scalar product of two vectors[1], *dlst* generates finite, descending lists of natural numbers.

$$
\begin{array}{ll}
\textit{sprod}\ \mathrm{L}\ [\ ] & := \mathrm{L}{=}[\ ] \to 0. \\
\textit{sprod}\ [\mathrm{H1}|\mathrm{T1}]\ [\mathrm{H2}|\mathrm{T2}] & := \mathrm{H1}{*}\mathrm{H2} + \textit{sprod}\ \mathrm{T1}\ \mathrm{T2}. \\
\textit{dlst}\ \mathrm{N} & := \mathrm{N}{>}0 \to [\mathrm{N}|\textit{dlst}(\mathrm{N}{-}1)]\ \Box\ [\ ]. \\
\textbf{solve}\ \textit{sprod}\ (\textit{dlst}\ 2)\ (\textit{dlst}\ 3). &
\end{array}
$$

Here, the unification of L and (*dlst* 2) does not force any evaluation of (*dlst* 2). L is just bound to it. In order to unify [ ] with the value of (*dlst* 3), (*dlst* 3) needs to be evaluated to head normal form. Hence, the rule for *dlst* is tried delivering the result [3|...]. Since this value is not unifiable with the corresponding formal argument [ ] on the left hand side of the first rule for *sprod*, backtracking to the last function call, in this case (*dlst* 3), occurs. Since there are no additional rules for *dlst*, the computation of (*dlst* 3) fails and is deleted. Backtracking goes on, and the second rule for *sprod* is tried. Then the first actual argument (*dlst* 2) is evaluated matching the formal paramenter [H1|T1] and, again, (*dlst* 3) is demanded and reevaluated. The computation proceeds several steps but fails finally.

Note that lazy evaluation causes the arguments to be evaluated after the function call for *sprod*. This means that alternative computations for the arguments are tried, before the next rule for *sprod* is attempted. Moreover, it implies that all computations for the arguments are forgotten, when the next rule is tried, and they need to be reevaluated for each rule. We can conclude that the naive lazy strategy is inefficient, since it leads to an arbitrary amount of re-evaluations of arguments.

This inefficiency is avoided, if the arguments are evaluated before the function call. This is only safely possible, if the evaluation of the arguments is demanded. Furthermore note

[1] We moved the restriction on the first argument of the first rule to the body in order to stress several disadvantages of naive lazy narrowing.

that we have incrementally evaluated the arguments. But ideally we would like to evaluate both arguments before the selection process of the rules, by using demandedness information. The operational notion "demandedness" must not be confused with the semantical notion "strictness". A function is strict in some argument, if the result is undefined provided that this argument is undefined. Strictness is used in lazy functional languages in order to decide which arguments are needed anyway and may hence be evaluated in advance, avoiding the construction of some intermediate representation of unevaluated expressions. In purely functional programming, this is valid since the whole computation fails, if some subexpression fails. In functional logic programming, a failing subexpression causes backtracking, and some alternative computation will be tried. Note that only one (part of an) argument can be demanded at a time. If its evaluation fails, all remaining arguments are not evaluated. If, however, the evaluation succeeds and the values of the currently evaluated arguments are unifiable with the corresponding patterns on the left hand side of a rule, then the next argument may be demanded and can be evaluated. Hence, this stepwise notion of demandedness allows several arguments to be evaluated in advance, and it takes into account the patterns on the left hand side and especially their dependencies. Below, this will explained in detail.

3.1 How to Avoid Re-evaluations

The inefficiency due to frequent re-evaluations of the arguments can be avoided, if the arguments are evaluated before the function call, more precisely: before a choice point for the function is created. In fact, this idea has already been followed in several approaches [MKLR90, HLW92, JMM92]. Note that this preliminary evaluation of the arguments is much more important for functional logic languages than for purely functional languages, where just the construction of an intermediate representation of unevaluated expressions is saved. However, the mentioned approaches have the drawback that they do not take into account dependencies between the arguments, but consider each argument separately. This leads to a blind evaluation of arguments even if it is already clear that no rule will be applicable, since no left hand side of a rule is unifiable with the values of the arguments computed up to now. Or, vice versa, the evaluation of some argument is delayed although demandedness is derivable.

The approaches presented in [HLW92] and [JMM92] for example end up with a notion of demandedness which leads to the evaluation of both arguments of the goal in the *sprod* example. The approach of [MKLR90] will evaluate the second argument.

If the demanded arguments are evaluated before the function call, this has an additional advantage. Since backtracking now first considers all the rules for f, before it tries alternative computations for the arguments, infinite computations due to infinitely many solutions of some argument expressions can be avoided. Consider the following rules (using: datatype $nat := zero \mid s\ nat.$):

$$one\ zero := s\ zero. \qquad\qquad one\ (s\ \mathrm{N}) := one\ \mathrm{N}.$$

The goal (one (one X)) leads to an infinite computation with the naive lazy strategy, since the first rule for one requires to find a solution of (one X) which is (unifiable with) $zero$. (one X) has infinitely many solutions, each binding X to a different natural number, but all of them fail to deliver the desired value $zero$. All of these infinitely many solutions are tried, and the second rule for one is never reached. If the demanded argument (one X) of the outer call to one is evaluated before the call to one, the second rule for one will be tried before a new solution for (*one* X) is tried. In the present example, the second rule for *one* leads to infinitely many solutions of the goal. Of course, it is possible to find examples where it is better to first try an alternative solution of an argument before trying the next rule. The point is that it is just more reasonable to explore a two dimensional search space, which has one finite dimension (the number of rules) and one possibly infinite dimension (the number of solutions of an expression), by first fixing a value in the second dimension and then scanning the finite dimension instead of vice versa.

Another problem is introduced, if indexing and unification are separated. The problem occurs, if a binding of a variable in the goal is postponed and some arguments which rely on this binding are executed in advance. Consider the goal: *foo* X (*loop* X) and the additional rules:

$$\textit{foo}\ 1\ 1 := 1. \qquad \textit{loop}\ 0 := \textit{loop}\ 0. \qquad \textit{loop}\ 1 := 1.$$

The first argument of *foo* is demanded, but it is already in hnf. Nevertheless, X needs to be bound to 1, before the second argument *loop* X is evaluated. Otherwise the search space for *loop* X is not yet restricted by the requirement $X = 1$, and the first rule for *loop* is applicable, leading to an infinite computation. Note that the binding of X has to be deleted, before the next rule for *foo* is considered. Otherwise different bindings for X, required by other rules, could not be tried.

3.2 Demand Patterns

The degree of demandedness of the arguments is expressed by a *demand pattern*. Since, in this paper, we want to focus on indexing, we will consider demand patterns which only reflect the demandedness due to the left hand sides of the rules. [JMM92] show how to use demandedness information corresponding to the right hand sides of the rules. This is in fact independent of the indexing scheme and can be added to the presented approach.

A *demand pattern* has the form

- $or(p_1,\ldots,p_n)$ (or-pattern) where p_i is a demand pattern $(1 \leq i \leq n,\ n > 0)$,
- $c(p_1,\ldots,p_n)$ (and-pattern) where p_i is a demand pattern $(1 \leq i \leq n,\ n \geq 0)$,
- *any*.

In the *sprod* example, we get the following demand pattern for the arguments of *sprod*:

$$tupel(or(any, [\ |\ ](or(any),\ or(any))),\ or([\ ], [\ |\ ](or(any), or(any))))$$

where *tupel* is an auxiliary constructor to combine the arguments of *sprod* and [|] is used in prefix instead of mixfix notation.

The demand patterns will be used by the indexing scheme in order to guide the selection of an applicable rule and the evaluation of the arguments. Remember that an indexing scheme for lazy narrowing needs to cope with both. If more information is needed to select a rule, additional evaluations of some parts of the arguments are incrementally initiated.

Since the demand patterns shall be used to select a rule, each and-pattern and any-pattern needs to be annotated with a set of rule numbers. The corresponding rules are possibly applicable, if this pattern is reached. For any-patterns, this set is (for technical reasons) the set of rules which are not mentioned in the other cases. An annotated pattern is represented by a pair $\langle p, R\rangle$, where p is an and- or any-pattern and R is the corresponding set of rule numbers. For the above example, we get the following annotated demand pattern:

$$\langle tupel(or(\langle any,\{1\}\rangle,\ \langle[\ |\ ](or(\langle any,\{2\}\rangle),\ or(\langle any,\{2\}\rangle)),\ \{2\}\rangle),$$
$$or(\langle[\ ],\{1\}\rangle,\ \langle[\ |\ ](or(\langle any,\{2\}\rangle),\ or(\langle any,\{2\}\rangle)),\ \{2\}\rangle)),\ \{1,2\}\rangle$$

Formally, the generation of annotated demand patterns can be described as follows:

$$generate(\{f\ t_{i,1}\dots t_{i,n}\ :=\ e_i.\ \ |\ \ 1\le i\le m\}) :=$$
$$\langle\ tupel(\ gen(\langle t_{1,1},\{1\}\rangle\dots\langle t_{m,1},\{m\}\rangle),\dots,\ gen(\langle t_{1,n},\{1\}\rangle\dots\langle t_{m,n},\{m\}\rangle)\ ,\ \ \{1,\dots,\mathrm{m}\}\rangle$$

$gen(\langle t_1,r_1\rangle\dots\langle t_n,r_n\rangle) :=$
if $\{t_1,\dots,t_n\}\cap Var=\emptyset$ then $or(\,aux(M_{c_1}),\dots,\ aux(M_{c_k}))$
else $or(\,aux(M_{c_1}),\dots,\ aux(M_{c_k}),\langle any,\ R\rangle)$
where $c_1,\dots,c_k$ are the topmost constructors of the terms $t_1,\dots,t_n$ and
$M_{c_i} := \{\langle t,r\rangle\ \mid\ \langle t,r\rangle\in\{\langle t_1,r_1\rangle,\dots,\langle t_n,r_n\rangle\},\ \ t=(c_i \text{ terms})\}, (1\le i\le k),$
$R := \{r\mid\langle X,r\rangle\ \mid\ \langle X,r\rangle\in\{\langle t_1,r_1\rangle,\dots,\langle t_n,r_n\rangle\},\ \ X\in Var\}$
$aux(\{\langle(c\ t_{1,1}\dots t_{1,n}),r_1\rangle,\dots,\langle(c\ t_{m,1}\dots t_{m,n}),r_m\rangle\}) :=$
$\langle c(gen(\langle t_{1,1},r_1\rangle\dots\langle t_{m,1},r_m\rangle),\dots,\ gen\ (\langle t_{1,n},r_1\rangle\dots\langle t_{m,n},r_m\rangle))\ ,\ \ \{r_1,\dots,r_m\}\rangle$

This algorithm ensures that each pattern has for each constructor at most one direct subpattern and that the direct subpatterns of or-patterns are and-patterns (or any-patterns) and vice versa. The latter property is used by the indexer in Section 4. Moreover, for an or-pattern $or(p_1,\dots,p_n)$, the annotations (i.e. the sets of possible rules) of different subpatterns p_i and p_j (for $i\neq j$) are disjoint.

4 An Indexing Scheme for Babel

Before we come to a detailed explanation of the indexing scheme, let us first discuss the main ideas using the *sprod* example.

The demand patterns are processed from left to right. In the example, we handle $or(\langle any,\{1\}\rangle,\ \langle[\ |\ ](or(\langle any,\{2\}\rangle),\ or(\langle any,\{2\}\rangle)),\ \{2\}\rangle)$ first. Here, the annotation $\{1\}$ of *any* contains a smaller rule number than the annotation$\{2\}$ of $[\ |\ ]$. This indicates that the rule which does not demand the evaluation of the first argument comes (textually) before the rule which does. According to our strategy to try the rules in their textual order, we are not allowed to evaluate this argument right now. After this, we consider the second argument of *sprod* and the corresponding pattern $or(\langle[\ ],\{1\}\rangle,\ \langle[\ |\ ](or(\langle any,\{2\}\rangle),$ $or(\langle any,\{2\}\rangle)),\ \{2\}\rangle)$. Since both rules are still possible, we have to check, whether we are allowed to evaluate the second argument. Here, both rules demand the evaluation to head normal form. If this evaluation delivers the head normal form $[\ ]$, we have to select rule 1 whereas the evaluation to $[\dots|\dots]$ forces rule 2 to be selected. If the result is a variable, both rules keep possible and will be tried in their textual order. After these preliminary explanations, we present the scheme in more detail.

4.1 Machine Instructions

We start with a brief description of the instructions used in the scheme. Most of them are well known from the WAM (Warren Abstract Machine [Wa83]), although we use the slightly different abstract machine BAM [Lo91, LW91] which has a (data)stack–based parameter passing mechanism rather than a register–based one.

TryMeElse l creates a new choice point on top of the runtime stack and stores the essential parameters of the current state. The assigned label l always points to a RetryMeElse or TrustMe instruction. The machine will jump to l, if the computation fails.

RetryMeElse l restores the previous state (backtracking). The topmost choice point is updated and points now to label l.

TrustMe removes the topmost choice point after the restoration of the previous state. All three instructions increment the program counter by one.

Try/Retry/Trust l work quite similar to the previous instructions. They differ only in the continuation behaviour: They store the address of the following instruction in the choice point and jump to l.

Switch l_{n+2} l_{n+1} $(c_1, l_1) : \ldots : (c_n, l_n)$ analyzes the topmost data stack entry and jumps depending on its value. It jumps to label l_{n+2}, if it is a variable, to l_{n+1} if it is a constructor not contained in the constructor list (default) which is specified by the third parameter. If it matches constructor c_i, it jumps to the label l_i.

Initiate checks the item pointed to by the topmost entry of the data stack and initiates its evaluation to head normal form.

BuildStructure c n creates a n-ary structure node in the heap and pushes the addresses of the components on top of the data stack.

Failure causes immediate backtracking, and

Jump l**/Pop** have the usual meaning.

4.2 Compilation Scheme

The previous discussion has shown that indexing is not only a useful mechanism to directly access the applicable rules at runtime but it can also initiate the evaluation of demanded arguments. An example which explains the interaction follows below. Let us now specify the compilation scheme formally. We choose a "top-down style", i.e. we begin with the translation of each function.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{functrans}(< f t_1^i \ldots t_n^i = e^i \mid 1 \leq i \leq r >, \mathrm{p}, \mathrm{l}) :=} \\
& \multicolumn{2}{l}{\text{if r=1 then } \mathit{ruletrans}(f t_1^1 \ldots t_n^1 = e^1, \mathrm{l}_1)} \\
& \multicolumn{2}{l}{\text{else}} \\
& & \mathit{indextrans}(1{:}\ldots{:}\mathrm{n}{:}\mathbf{b}, \{1,\ldots,\mathrm{r}\}, \{1,\ldots,\mathrm{r}\}, (\mathrm{p},\emptyset), \mathrm{l}_0, \mathrm{l}_1{:}\ldots{:}\mathrm{l}_r) \\
& \mathrm{l}_1 & \mathit{ruletrans}(f t_1^1 \ldots t_n^1 = e^1, \mathrm{l}_1) \\
& & \vdots \\
& \mathrm{l}_r & \mathit{ruletrans}(f t_1^r \ldots t_n^r = e^r, \mathrm{l}_r)
\end{array}
$$

Clearly, if the function consists of one rule indexing makes no sense. Otherwise, the indexing code precedes the usual code of the rules. It is produced by *indextrans* taking 6 arguments for control:

- The first argument lists the numbers of argument positions of the function or components of structures. This sequence contains a mark **b** (boundary) behaving as a delimiter which initiates pop operations (see below). Initially, the sequence contains the argument positions in left to right order.

- Next the set, actually a subset, of possibly applicable rules (identified by their numbers) is given. The initial value contains all rules.

- Also the third parameter denotes a set of rules. Whenever bindings are done this parameter will be modified in order to store the rules which remain applicable and respect these bindings. Without such a set the indexer would forget during subsequent steps that it has done these bindings and would compute a superset of the valid rules.

- Whenever the indexer looks inside a structure, e.g. s(0) inside c(X,Y,s(0),Z), in order to derive further restrictions of applicable rules, the visible rules are temporarily restricted to the ones being suitable for this structure. Other rules with more general terms like X or c(X,Y,U,V) are not under investigation but remain valid. The current environment consisting of the current pattern and these remaining rules is passed as the fourth parameter. It will be restored when returning from the deeper level. Initially the passed pattern p of *functrans* and the empty set are pushed.

- The next parameter is used for technical reasons and ensures different symbolic labels, and

- the last one stores the entry addresses of the rules as a sorted list which is passed through until all positions are processed.

The code of every rule is produced by the *ruletrans* scheme, which is omitted here due to the lack of space (see [Lo91] for details). In contrast to our implementation, we have not merged indexing and unification code here for simplicity of presentation. Indexing can be seen as a prepass over the arguments just before the execution of a rule starts.

$indextrans(\mathcal{P},\mathcal{R},\mathcal{S},(\mathrm{p},\mathcal{V}){:}\mathcal{ST},\mathrm{l},\mathcal{L})$:=
  if $(\mathcal{R} \cup \mathcal{V}) \cap \mathcal{S} = \emptyset$ then **Failure**
  else if $\mathcal{P}=\varepsilon$ then $build\_try\_trust((\mathcal{R} \cup \mathcal{V}) \cap \mathcal{S},\mathcal{L})$
  else let $\mathcal{P}=\mathrm{p}{:}\mathcal{P}'$ in
    if p=**b** then $indextrans(\mathcal{P}',\mathcal{R} \cup \mathcal{V},\mathcal{S},\mathcal{ST},\mathrm{l},\mathcal{L})$
    else if p=i then
      let $\mathrm{p}'=project(\mathrm{p},\mathrm{i})$ in
        if $demanded(\mathrm{p}',(\mathcal{R} \cup \mathcal{V}) \cap \mathcal{S})$ then
          Initiate
          $switchtrans(\mathrm{p}',\mathcal{P}',\mathcal{R},\mathcal{S},(\mathrm{p},\mathcal{V}){:}\mathcal{ST},\mathrm{l}_1,\mathcal{L})$
          $indexbindtrans(\mathrm{p}',\mathcal{P}',\mathcal{R},\mathcal{S},(\mathrm{p},\mathcal{V}){:}\mathcal{ST},\mathrm{l}_{1,n+2},\mathcal{L})$
        else
          $switchtrans(\mathrm{p}',\mathcal{P}',\mathcal{R},\mathcal{S},(\mathrm{p},\mathcal{V}){:}\mathcal{ST},\mathrm{l}_2,\mathcal{L})$
          Pop
          $indextrans(\mathcal{P}',\mathcal{R},\mathcal{S}, (\mathrm{p},\mathcal{V}){:}\mathcal{ST},\mathrm{l}_{2,n+2},\mathcal{L})$

The important part of the specification is *indextrans*. First it checks whether some of the currently considered rules are applicable (i.e. $(\mathcal{R} \cup \mathcal{V}) \cap \mathcal{S} = \emptyset$). If no applicable rule remains, an instruction is generated which initiates backtracking. Otherwise if all arguments and inner structure positions have been processed ($\mathcal{P} = \varepsilon$), the code chaining the rules is generated by the *build_try_trust* scheme.

If further analysis is possible and the delimiter **b** is reached (denoting that a structure has been completely considered), one entry of the "environment" stack is popped in order to analyze the upper level. Note that additional rule numbers are added to the set of applicable rules. These rules correspond to any–patterns, i.e. variables, encountered while traversing nested structures.

If none of the previous checks was successful, the code generation for one indexing step is possible. First, the demandedness of the argument position is checked. If it is demanded, the instruction Initiate is generated. *switchtrans* manages nonvariable (hnf-)expressions and *indexbindtrans* handles encountered variables, which must now be bound to retain safety with respect to laziness. Otherwise, the expression is analyzed without initiating any evaluation; variables are simply popped.

switchtrans(p,$\mathcal{P}$,$\mathcal{R}$,$\mathcal{S}$,$\mathcal{ST}$,l,$\mathcal{L}$):=
  let ($\langle$_,$\mathcal{A}_v\rangle$,p_1:...:p_n)=*split*(p) and p_i=$\langle c^i(p^i_1 \ldots p^i_{n_i})$,_$\rangle$) in
    Switch l_{n+2} l_{n+1} ((c^1,l_1):...:(c^n,l_n))
  l_1 *indexstructtrans*(p_1,$\mathcal{A}_v$,$\mathcal{P}$,$\mathcal{R}$,$\mathcal{S}$,$\mathcal{ST}$,l_1,$\mathcal{L}$)
    ⋮
  l_n *indexstructtrans*(p_n,$\mathcal{A}_v$,$\mathcal{P}$,$\mathcal{R}$,$\mathcal{S}$,$\mathcal{ST}$,l_n,$\mathcal{L}$)
  l_{n+1} *indextrans*($\mathcal{P}$,$\mathcal{R} \cap \mathcal{A}_v$,$\mathcal{S}$,$\mathcal{ST}$,$l_{n+1}$,$\mathcal{L}$)
  l_{n+2} ...

indexstructtrans($\langle c(p_1 \ldots p_n),\mathcal{A}\rangle$,$\mathcal{A}_v$,$\mathcal{P}$,$\mathcal{R}$,$\mathcal{S}$,(p,$\mathcal{V}$):$\mathcal{ST}$,l,$\mathcal{L}$):=
  indextrans(1:...:n:**b**:$\mathcal{P}$,$\mathcal{R} \cap \mathcal{A}$,$\mathcal{S}$,($\langle c(p_1 \ldots p_n),\mathcal{A}\rangle$,$\mathcal{V} \cup (\mathcal{A}_v \cap \mathcal{R})$):(p,$\mathcal{V}$):$\mathcal{ST}$,l,$\mathcal{L}$)

indexbindtrans(p′,$\mathcal{P}$,$\mathcal{R}$,$\mathcal{S}$,(p,$\mathcal{V}$):$\mathcal{ST}$,l,$\mathcal{L}$):=
  let $(\mathcal{R} \cup \mathcal{V}) \cap \mathcal{S} = \{r_1, \ldots, r_n\}$ and $r_i < r_{i+1}$ in
    if n=1 then
      buildtrans(p′,r_1)
      indextrans($\mathcal{P}$,$\mathcal{R}$,$\{r_1\}$,(p,$\mathcal{V}$):$\mathcal{ST}$,l_1,$\mathcal{L}$)
    else
      ⋮
    l_i **TryMeElse** l_2//**RetryMeElse** l_{i+1}//**TrustMe**
      buildtrans(p′,r_i)
      indextrans($\mathcal{P}$,$\mathcal{R}$,$\{r_i\}$,(p,$\mathcal{V}$):$\mathcal{ST}$,l_i,$\mathcal{L}$)
      ⋮

The *switchtrans* scheme generates code which expects an expression in hnf on top of the data stack and branches depending on its value. The passed or–pattern is split into one any–pattern and one or–pattern for each occurring top level constructor[2]. The current context is saved consisting of the current pattern and the valid rules (cf. *indexstructtrans*). The (potentially) valid rules are denoted by the any–pattern (restricted by $\mathcal{R}$ and joined with the rules $\mathcal{V}$ inherited from the upper levels of the visited structure). The list of positions

[2]For simplicity it is assumed that each constructor has a unique arity.

which must be processed is extended by the positions of the components of the considered structure and terminates with a delimiter.

In the default case, the set of applicable rules can be restricted in a similar way. The code at label l_{n+2} is the code following the code produced by *switchtrans*.

Let us now discuss the demand-context: *indexbindtrans* generates code which builds the term (*buildtrans*) the variable must be bound to. Due to the possibility of different bindings reflecting the alternative computation paths, a choice point may become necessary chaining the alternatives. The further analysis is performed by recursive calls to *indextrans*. The rule that causes the binding is noted and must remain valid during the following process.

Note that a simple optimization can check contiguous rules on variant terms at the current position and can group them. We restrict this grouping to contiguous rules (omitting the invalid ones) since we respect the textual ordering of the rules.

buildtrans($\langle$*any*,$\mathcal{A}\rangle$,k) := **Pop**
buildtrans($\langle$c($p_1 \ldots p_n$),$\mathcal{A}\rangle$,k) :=
    BuildStructure c n
    buildtrans(p_1,k)
    $\vdots$
    buildtrans(p_n,k)
buildtrans(or($\langle p_1, \mathcal{A}_1\rangle$,...,$\langle p_n, \mathcal{A}_n\rangle$),k) :=
    if $\exists i_0 : k \in \mathcal{A}_{i_0}$ then *buildtrans*($\langle p_{i_0}, \mathcal{A}_{i_0}\rangle$,k)
    else **Pop**

build_try_trust($\{r_1, \ldots, r_n\}, l_1 : \ldots : l_m$):=
    if n=1 then **Jump** l_{r_1}
    else let $r_i < r_{i+1}$ in
        Try l_{r_1}
        $\vdots$
        Retry l_{r_i}
        $\vdots$
        Trust l_{r_n}

Structures are created by the instruction BuildStructure behaving similar to a unification instruction in write mode. It puts a structure tag together with the arguments into the heap. Additionally, pointers to the components are pushed onto the data stack. A variable (inside the term) is popped.

project($\langle$c($p_1 \ldots p_n$),$\mathcal{A}\rangle$, i) := p_i

split(or(p_1,...,p_n)) :=
    if $\exists i_0 : p_{i_0} \in \{\text{any}\} \times \mathcal{P}(\mathbb{N})$ then
        (p_{i_0},p_1:...:p_{i_0-1}:p_{i_0+1}:...:p_n)
    else ($\langle$any$_1$,$\emptyset\rangle$,p_1:...:p_n)

demanded(or(p_1,...,p_n),$\mathcal{R}$) :=
    let p_i=(_,$\mathcal{A}_i$) and j=min $\mathcal{R}$ in
        if $\exists i_0 : j \in \mathcal{A}_{i_0} \wedge p_{i_0} \notin \{\text{any}\} \times \mathcal{P}(\mathbb{N})$
        then true
        else false

The *indextrans* and the *switchtrans* schemes access auxiliary functions: *project*, *split*, and *demanded*. *project* extracts components from an and–pattern and *split* distinguishes between any– and and–patterns considering the first level of the input or–pattern. *demanded* checks whether the first applicable rule (in textual order) does not have an any–pattern at the considered position.

4.3 An Example

The following table shows the indexing code for the *sprod* example. We assume that both arguments of *sprod*, are pushed onto the data stack.

1:	Switch	101 96 ([|],2)
2:	Switch	49 3 ε
3:	Switch	26 4 ε
4:	Initiate	
5:	Switch	18 17 ([ ],6):([|],7)
6:	Jump	rule_1
7:	Switch	12 8 ε
8:	Switch	10 9 ε
9:	Jump	rule_2
10:	Pop	
11:	Jump	rule_2
12:	Pop	
13–16:	updated copy of ll. 8–11	
17:	Failure	
18:	TryMeElse	21
19:	BuildStructure	[ ] 0
20:	Jump	rule_1
21:	TrustMe	
22:	BuildStructure	[|] 2
23:	Pop	
24:	Pop	
25:	Jump	rule_2
26:	Pop	
27–48:	updated copy of ll. 4–25	
49:	Pop	
50–95:	updated copy of ll. 3–48	
96:	Switch	99 98 ([ ],97)
97:	Jump	rule_1
98:	Failure	
99:	BuildStructure	[ ] 0
100:	Jump	rule_1
101:	Pop	
102–123:	updated copy of ll. 4–25	

rule_1: *ruletrans(sprod* L [ ]:= L=[ ] $\rightarrow$ 0,rule_1)

rule_2: *ruletrans(sprod* [H1|T1] [H2|T2] := H1*H2 + (sprod T1 T2),rule_2)

Code sharing decreases the number of instructions, e.g. the simple replacement of 'updated copy of ll. L1–L2' by 'Jump L1' will reduce the code size enormously.

5 Machine Implementation and Experimental Results

The following example programs were executed on a stack-based narrowing machine [Lo91]. The narrowing machine is an extension of a pure reduction machine enriched by mechanisms for performing unification and backtracking similar to the Warren Abstract Machine [Wa83].

The current implementation of the indexing module incorporates several optimizations, which have been omitted due to the lack of space (see [HKW93] for details). These optimizations include the prevention of code duplication. Moreover, the code for unification and indexing is merged. All these improvements are straightforward. A further optimization is possible by using knowledge about modes. Such a mode analyser for lazy narrowing is currently under investigation. However, our examples show that code explosion is in practice not as harmful as it seems to be (see Table 1).

The naive lazy scheme and the new approach have been compared for several example programs. We have tried both approaches with and without another refinement which prevents the deletion of deterministic computations when backtracking. This also avoids some superfluous re-evaluations (see [HLW92] for details).

The example programs include quicksort with a list of n elements with initially reverse ordering, the computation of all sublists of reverse $[1 \ldots n]$, the n-queens problem with symbolic representation of lists and natural numbers, the satisfiability problem (3-SAT) with n variables and 200 terms, the n-th symbolic derivation of a polynomial, the computation of all prime numbers up to n, and a Babel parser using higher order functions. For comparison purposes we have also listed the runtimes for eager narrowing. An eager strategy yields better runtimes for some example programs, but a lazy strategy allows, of course, a more flexible and powerful way of programming [Hu84]. The runtimes in Table 1 show a con-

			without refinement		with refinement		code size
example	n	Eager	Naive Lazy	with Idxg	Naive Lazy	with Idxg	factor
quicksort	50	0.4333	$\gg$ 1000.0	0.683	1.513	0.677	1.06
sublists	10	0.400	6.663	0.820	1.003	0.820	1.37
queens	7	11.800	45.710	18.003	36.350	17.850	1.51
sat	3	5.247	$\gg$ 1000.0	0.250	5.096	0.237	1.32
derivation	6	1.763	$\gg$ 1000.0	1.930	3.467	1.930	1.18
sieve	1000	3.180	$\gg$ 1000.0	4.950	8.440	4.930	1.11
parser		1.330	2.886	2.153	2.150	1.867	1.16
population		0.243	0.900	0.289	0.900	0.280	1.21

Table 1: Runtimes of the example programs in seconds and increase of code size for indexing.

siderable advantage of the new approach due to the prevention of multiple re-evaluations. Note that in some cases for the naive strategy the programs do not stop in an acceptable period of time.

Note that the new approach leads to a considerable improvement of runtimes even if other optimizations are added which prevent some (successful) re-evaluations of deterministic computations. The best runtimes are reached by the combination of these techniques.

Some remarkable results are observed for the 3-SAT program. The new approach causes a high speed up also in comparison to the eager evaluation strategy. This is due to the combination of the two following effects and the big search space spawned by the problem. On the one hand, superfluous computations can be avoided by using a lazy strategy whereas on the other hand multiple re-evaluations of arguments caused by the naive lazy strategy are eliminated. Experiments have shown that even the naive strategy combined with a static demandedness analysis as proposed in [HLW92] leads to multiple re-evaluations and worse runtimes. Moreover notice that we have renounced to show typical "database applications" (except the population example) in order to avoid the impression that we have used custom tailored examples.

6 Conclusions and Future Work

A new full indexing scheme for lazy narrowing has been presented that incorporates several advantages in comparison to former approaches. Firstly, the new technique is more lazy than in the approaches introduced before, because it takes into account the dependencies of the arguments and their specific instantiations. Secondly, the integration of indexing and variable bindings guarantees that the search space for further argument computations within the unification process is decreased. A further essential advantage is given by the early evaluation of demanded arguments that prevents re-evaluations and leads to dramatically improved runtimes. It has also been shown that a combination with other optimization techniques, e.g. a refinement of backtracking, is possible.

In the future, we will extend our scheme to a notion of demandedness which takes into account the right hand sides of the rules.

References

[AK91] H. Ait-Kaci: Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, 1991.

[BL86] M. Bellia, G. Levi: The Relation between Logic and Functional Languages, Journal of Logic Programming, Vol.3, 1986, 217-236.

[DL86] D. DeGroot, G. Lindstrom (eds.): *Logic Programming: Functions, Relations, Equations*, Prentice Hall, 1986.

[Ha92] W. Hans: A Complete Indexing Scheme for WAM-based Abstract Machines, Procs. PLILP'92, LNCS 631, 1992, 232-244.

[HKW93] W. Hans, H. Kuchen, S. Winkler: Advanced Indexing Techniques for the Functional Logic Programming Language BABEL, Technical Report, to appear.

[HLW92] W. Hans, R. Loogen, S. Winkler: On the Interaction of Lazy Evaluation and Backtracking, Procs. PLILP'92, LNCS 631, 1992, 355-369.

[HM89] T. Hickey, S. Mudambi: Global Compilation of Prolog, Journal of Logic Programming, Vol. 7, 1989.

[Hu84] J. Hughes: Why Functional Programming Matters, Technical Report 16, Programming Methodology Group, University of Goteborg, November 1984.

[JMM92] J.A. Jiménez-Martín, J. Mariño-Carballo, J.J. Moreno-Navarro: Some Techniques for the Efficient Compilation of Lazy Narrowing into Prolog, Procs. LOPSTR'92, Manchester, 1992.

[KS88] S. Kliger, E. Shapiro: A Decision Tree Compilation Algorithm for FCP(–,;,?), Procs. 5th ICSLP, MIT Press, 1988.

[KT91] M. Korsloot, E. Tick: Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages, Procs. 8th ICSLP, MIT Press, 1991.

[KLMR92] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-Narrowing to Implement a Functional Logic Language, Tech. Report, Univ. Politecnica Madrid, 1992.

[Mi78] R. Milner: A Theory of Type Polymorphism in Programming, JCSS 17(3), 1978, 348-375.

[Lo91] R. Loogen: From Reduction Machines to Narrowing Machines, TAPSOFT, LNCS 494, 1991, 438-457.

[LW91] R. Loogen, S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages, Procs. PLILP'91, LNCS 528, 1991, 335-346.

[MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, Procs. ALP'90, LNCS 463, 1990, 298-317.

[MR92] J.J.Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language Babel, J. Logic Programming, 12, 1992, 189-223.

[PN91] D. Palmer, L. Naish: NUA-Prolog: An Extension to the WAM for Parallel Andorra, Procs. 8th ICSLP, MIT Press, 1991.

[Re85] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages, Procs. Int. Symp. on Logic Programming, IEEE Comp. Soc. Press 1985, 138-151.

[Wa83] D.H.D. Warren: An Abstract Instruction Set, Tech. Note 309, SRI International, Menlo Park, 1983.