Proc. Algebraic and Logic Progr. 90, LNCS 463, Springer Verlag 1992, 298–317

Lazy Narrowing in a Graph Machine*

Juan José Moreno-Navarro
Universidad Politécnica de Madrid†

Herbert Kuchen, Rita Loogen
RWTH Aachen‡

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid§

Abstract

The paper investigates the implementation of lazy narrowing in the framework of a graph reduction machine. By extending an appropriate architecture for purely functional languages an abstract graph narrowing machine for a functional logic language is constructed. The machine is capable of performing unification and backtracking. The techniques used in functional languages to cope with lazy evaluation are not directly applicable, but must be modified due to the logic component of the implemented language. A prototype implementation of the new machine has been developed.

1 Introduction

During the last years, several approaches have been proposed to achieve an integration of functional and logic programming languages in order to combine the advantages of the two main declarative programming paradigms in a single framework [DeGroot, Lindstrom 86], [Bellia, Levi 86]. The so called *functional logic languages* [Reddy 85,87] retain functional syntax but use *narrowing* – a unification based parameter passing mechanism which subsumes reduction and SLD-resolution – as operational semantics.

We investigate in this paper a *lazy implementation* of the functional logic language BABEL [Moreno, Rodríguez 89] on a graph narrowing abstract machine. Originally, BABEL was designed as a first order, untyped language. In a recent paper [Kuchen et al. 90] we extended the language by higher order functions and polymorphic types, and we developed an eager implementation by extending the sequential kernel of a parallel (programmed) graph reduction machine which had been designed for the execution of functional programming languages [Loogen et al. 89] by logic features, namely unification and backtracking. In this paper, we modify the previous machine to support a lazy evaluation strategy. For simplicity, we restrict ourselves to the first order, untyped subset of BABEL. As we shall sketch later, our approach can be easily extended to deal with higher order functions.

We show that the way in which a functional implementation treats lazy evaluation is not quite adequate for a functional logic language. It is advisable to use a special class of so-called *uniform programs* which allow a more appropriate implementation. An automatic transformation of general BABEL programs into uniform programs is described.

The paper is organized as follows. Section 2 gives a short description of the language BABEL. Uniform BABEL programs are discussed in section 3. Section 4 explains the structure of the abstract

*This work has been partially supported by the german-spanish cooperation action HA 24/B and by the spanish projects PRONTIC TIC 89/0104 and Precompetitivo UPM A-90002001/44.

†Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain.

‡Lehrstuhl für Informatik II, Ahornstraße 55, D-5100 Aachen, Germany.

§Departamento de Informática y Automática, Facultad de C.C. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain.

machine, the evaluation mechanism, the effects of machine instructions and the code generation schemes. In section 5, we sketch how our approach could be extended to higher order BABEL. Some remarks about a prototype implementation are collected in section 6. Finally, section 7 includes some conclusions and comments on related work.
An extended version of this paper, including detailed formal specifications, is available as technical report [Moreno et al. 90].

2 The Functional Logic Language BABEL

The first-order core of the language BABEL was designed to achieve integration of functional and logic programming in a flexible and mathematically well-founded way [Moreno, Rodríguez 88,89], [Moreno 89]. It is a functional logic language [Reddy 85,87] that is based on a constructor discipline and uses narrowing as evaluation mechanism.

2.1 Syntax of First-order BABEL

Let $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ be ranked alphabets of *constructors* and *function symbols* respectively. We assume that these alphabets contain some *primitive symbols*, including at least the nullary constructors 'true' and 'false', the usual boolean operators, an equality operator and conditional operators. In the following, letters $a, b, c \ldots$ are used for constructors and the letters $f, g, h \ldots$ for function symbols.
The following syntactic domains are distinguished:

- *Variables* $X, Y, Z \ldots \in Var$
- *Terms* $s, t, \ldots \in Term$: $\quad t ::= X \quad$ % Variable $\quad \mid (c\, t_1 \ldots t_n) \quad$ % $c \in DC^n, n \geq 0$
- *Expressions* $M, N \ldots \in Exp$: $\quad M ::= t \quad$ % $t \in Term$ $\quad \mid (c\ M_1 \ldots M_n) \quad$ % Construction, $c \in DC^n, n \geq 0$ $\quad \mid (f\ M_1 \ldots M_n) \quad$ % Application, $f \in FS^n, n \geq 0$

In particular, we get expressions built from the primitive function symbols, which are denoted by prefix, infix and mixfix operators:

$\neg B$	% negation
(B_1, B_2)	% sequential conjunction
$(B_1; B_2)$	% sequential disjunction
$(B \rightarrow M)$	% guarded expression: **if** B **then** M **else** undefined
$(B \rightarrow M_1 \Box M_2)$	% conditional expression: **if** B **then** M_1 **else** M_2
$(M_1 = M_2)$	% weak equality (meaning explained later)

When writing concrete expressions, we keep only those parentheses needed to avoid ambiguity.
A first-order BABEL-*program* consists of a set of defining rules for the non predefined symbols in FS. The rules for the predefined symbols are implicitly added to every program, and will be presented later.
Let $f \in FS^n$. Each defining *rule* for f must have the form

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

and satisfy the following restrictions:

1. *Data Patterns*: $t_i \in Term$.

2. *Left Linearity*: $f\ t_1 \dots t_n$ does not contain multiple variable occurrences.

3. *Restrictions on free variables*: Variables occurring in the rhs but not in the lhs are called *free*. Occurrences of free variables are allowed in the guard, but not in the body.

4. *Nonambiguity*: Given any two rules for the same function symbol f:

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

   one of the three following cases must hold:

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

   (b) *Fusion of bodies*: $f\ t_1 \dots t_n$ and $f\ s_1 \dots s_n$ have a most general unifier (m.g.u.) σ such that $M\sigma, N\sigma$ are identical[1].

   (c) *Incompatibility of guards*: $f\ t_1 \dots t_n$ and $f\ s_1 \dots s_n$ have a m.g.u. σ such that the conjunction $(B, C)\sigma$ is incoherent.

Condition (c) needs an additional explanation. *Incoherence* must be defined as a decidable syntactical property of expressions, and chosen in such a way that no incoherent expression may denote the boolean value 'true'. For a particular choice of this notion, cfr. [Moreno, Rodríguez 89].

2.2 Predefined Rules

Since BABEL is a lazy language, primitive functions can be specified by the same kind of rules as user defined functions. We assume that the following *predefined rules* belong to any program.

- Rules for the boolean operations

$$\boxed{\begin{array}{lcl} \neg\,\text{false} & := & \text{true} \\ \neg\,\text{true} & := & \text{false} \end{array}} \qquad \boxed{\begin{array}{lcl} \text{false, Y} & := & \text{false} \\ \text{true, Y} & := & \text{Y} \end{array}} \qquad \boxed{\begin{array}{lcl} \text{false; Y} & := & \text{Y} \\ \text{true; Y} & := & \text{true} \end{array}}$$

- Rules for weak equality

$$\boxed{\begin{array}{l} (c = c) := \text{true} \qquad\qquad\qquad\qquad \%\ c \in DC^0,\ \text{constant} \\ ((c\,X_1 \dots X_n) = (c\,Y_1 \dots Y_n)) := (X_1 = Y_1), \dots, (X_n = Y_n) \quad \%\ c \in DC^n \\ ((c\,X_1 \dots X_n) = (d\,Y_1 \dots Y_m)) := \text{false} \quad \%\ c \in DC^n,\ d \in DC^m\ \text{different} \end{array}}$$

- Rules for guarded and conditional expressions

$$\boxed{(\text{true} \rightarrow X) := X} \qquad \boxed{\begin{array}{ll} (\text{true} \rightarrow X \Box Y) & := X \\ (\text{false} \rightarrow X \Box Y) & := Y \end{array}}$$

The rules for conjunction (,) and disjunction (;) reflect the sequential character of these operations. The rules for weak equality specify that an expression $(M_1 = M_2)$ will evaluate to true if M_1, M_2 evaluate both to the same finite term, and will evaluate to false if M_1, M_2 evaluate to different terms. BABEL supports infinite terms through lazy evaluation. Thus, an expression $(M_1 = M_2)$ may be lazily evaluated to false even if the complete evaluation of M_1 or M_2 would not terminate.
Other useful primitives (e.g. arithmetic operations, list operations, etc.) could be introduced easily by predefined rules.

[1] As usual, $M\,\sigma$ denotes the expression M where all variables are replaced according to σ.

2.3 Programming in BABEL

The fact that BABEL supports functional programming style is apparent. Moreover, it embeds pure PROLOG, since Horn clauses can be expressed as guarded rules with 'true' as body.
We show here a small example program which illustrates several features: Use of lazy functions, simulation of Horn clauses and computation of multiple solutions for a given goal. The program solves the problem of computing Fibonacci numbers, using a lazy list. We adopt PROLOG-like notation for lists and numbers.

/* Function to construct the infinite sequence of Fibonacci numbers */
$\mathit{fib\_nbs}\ X\ Y := [X \mid \mathit{fib\_nbs}\ Y\ (X+Y)].$

/* Function to calculate the nth element of a list */
$\mathit{nth}\ 0\ [X \mid Xs] := X.$
$\mathit{nth}\ (\mathit{suc}\ N)\ [X \mid Xs] := \mathit{nth}\ N\ Xs.$

/* Predicate to decide if X is a Fibonacci number */
$\mathit{fib\_nb}\ X := ((\mathit{nth}\ N\ (\mathit{fib\_nbs}\ 1\ 2)) = X) \rightarrow \mathit{true}.$

/* Rules for $+$, *less* should be added */

/* Calculate a Fibonacci number less than 5 */
solve $(\mathit{less}\ X\ 5), (\mathit{fib\_nb}\ X).$

$\triangleright$ true {X = 1}
$\triangleright$ true {X = 2}
$\triangleright$ true {X = 3}

2.4 Narrowing Semantics

In general, a *goal* for a given BABEL program is any expression M. To solve a goal, the BABEL machine tries to reduce it to a normalized form by means of *narrowing*. This means that the lhs of rules for the defined and predefined function symbols are unified with appropriate subexpressions, which are then replaced by the corresponding instance of the rule's rhs. This process is repeated until a normal form N is reached. Then, N is taken as the *result* of the evaluation, and all bindings of variables occurring in M that have been accumulated during the reduction are regarded as the *answer*, similarly as in PROLOG. The combination of result and answer is the *outcome* of the computation.

BABEL works with a *lazy* narrowing strategy. Hence, it tries to narrow expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the pattern in the lhs of some rule) and contributes to some later narrowing step at an outer position. Narrowing as the operational semantics of functional logic languages was introduced in [Reddy 85,87]. Now, we sketch a formal definition of lazy narrowing.

Narrowing rule

The narrowing rule describes how to apply a BABEL rule through unification.
We have

$$f\ M_1 \dots M_n \longrightarrow_{\sigma_{out}} R\sigma_{in}$$

if there is some variant $f\ t_1 \dots t_n := R$ of a rule in the program which shares no variables with $f\ M_1 \dots M_n$ and is such that $f\ t_1 \dots t_n$ and $f\ M_1 \dots M_n$ are unifiable with m.g.u. $\sigma = \sigma_{in} \cup \sigma_{out}$ where σ_{in} and σ_{out} record the binding of variables in $f\ t_1 \dots t_n$ and $f\ M_1 \dots M_n$, respectively.

One step narrowing relation

"*M narrows in one step* to N with answer substitution σ_{out}", in symbols

$$M \Longrightarrow_{\sigma_{out}} N$$

is specified as follows:

- Narrowing arguments in a construction:

$$\frac{M_i \Longrightarrow_{\sigma_{out}} N_i}{c\, M_1 \ldots M_i \ldots M_n \Longrightarrow_{\sigma_{out}} c\,(M_1\sigma_{out}) \ldots N_i \ldots (M_n\sigma_{out})} \quad \begin{pmatrix} c \in DC^n \\ 1 \le i \le n \end{pmatrix}$$

- Outermost narrowing by application of a rule:

$$\frac{f\, M_1 \ldots M_n \longrightarrow_{\sigma_{out}} N}{f\, M_1 \ldots M_n \Longrightarrow_{\sigma_{out}} N} \quad (f \in FS^n)$$

- Inner narrowing step (allowed only if it is demanded and contributes to some later narrowing at an outer position):

$$\frac{M_i \Longrightarrow_{\sigma_{out}} N}{f\, M_1 \ldots M_i \ldots M_n \Longrightarrow_{\sigma_{out}} f\,(M_1\sigma_{out}) \ldots N \ldots (M_n\sigma_{out})}$$

Narrowing relation

"*M narrows in several steps* to N with answer substitution σ_{out}", in symbols

$$M \stackrel{*}{\Longrightarrow}_{\sigma_{out}} N$$

is defined as the reflexive, transitive closure of the one step narrowing relation with composition of the substitutions. If $M \stackrel{*}{\Longrightarrow}_{\sigma_{out}} N$ we speak of a *lazy narrowing* computation with *answer* σ_{out} and (partially evaluated) *result* N. Performing lazy narrowing with a BABEL goal expression M may lead to several situations:

- *Success:* $M \stackrel{*}{\Longrightarrow}_{\sigma} t$ with $t \in Term$
- *Failure:* $M \stackrel{*}{\Longrightarrow}_{\sigma} N$, $N \notin Term$ and N is not further narrowable
- *Nontermination.* In this case we may still have $M \stackrel{*}{\Longrightarrow}_{\sigma_{out}} N$ where $N \notin Term$ and N's constructors give a partial result.

A more detailed presentation of BABEL's operational semantics is given in [Moreno, Rodríguez 89], where the reader will also find a discussion of declarative semantics and soundness and completeness issues.

3 Uniform BABEL Programs

In a lazy functional language, the reduction of an expression $f\ M_1 \ldots M_n$ by means of defining rules $f\ t_1 \ldots t_n := R$ would follow the steps:

1. The arguments M_i are evaluated as much as necessary to match them with the patterns t_i.
2. If matching succeeds, the rhs R is evaluated.
3. If matching fails, the next rule is tried in the same manner.

In order to implement BABEL's lazy narrowing, we must take into account that:

- Pattern-matching must be replaced by unification.
- If an argument does not unify with the corresponding term on the lhs of the rule, it does not imply that the rule is not applicable. Another evaluation of the argument could give a new result that unifies with the term.

Another problem is illustrated by the example program

$$\begin{array}{llll} \Pi_0: & R_1: & f\ 0\ \ 0 & := 0. \\ & R_2: & f(s\ X)\ 0 & := 1. \\ & R_3: & f\ X\ (s(s\ Y)) & := 2. \end{array}$$

and the expression $(f\ (f\ X\ Y)\ Z)$. Rule R_3 can be applied, while the rules R_1 and R_2 are *suspended*, i.e. they cannot be applied yet, since the first argument $(f\ X\ Y)$ is not enough evaluated, but they might become applicable later. All the rules can be used to reduce the subexpression $(f\ X\ Y)$.
The example shows that for an implementation *backtracking points due to different redexes* are needed (in addition to backtracking points due to different applicable rules, which are of course also required). This leads to complicate, inefficient and possibly non-complete implementations. [Echahed 88], while investigating complete narrowing strategies for term rewriting systems, points out this kind of difficulties and shows how to avoid them in some cases by transforming the given term rewriting system. The key idea is that *backtracking due to different redexes* is replaced by *backtracking due to different rules.*

3.1 Non–subunifiable BABEL Programs

Our implementation is based on an automatic transformation of BABEL programs to programs which fulfil certain syntactical restrictions.
Two lhs's $f\ t_1 \ldots t_n$ and $f\ s_1 \ldots s_n$ are said to be *subunifiable*, if there exists an i $(1 \leq i \leq n)$ such that t_i and s_i are subunifiable. Two linear terms t and s are *subunifiable*, if:

- t is a variable and s is a non-variable term, or
- $t = (c\ t_1 \ldots t_n)$ and $s = (c\ s_1 \ldots s_n)$ for some constructor $c \in DC^n$ and terms $t_1, \ldots, t_n$, $s_1, \ldots, s_n$ $(n \geq 1)$, and there exists an i $(1 \leq i \leq n)$ such that t_i and s_i are subunifiable.

A *non-subunifiable* BABEL program is any BABEL program, where the lhs's of the rules are pairwise not subunifiable.

It is easy to prove that for a non-subunifiable BABEL program there is no expression M for which there are applicable and suspended rules at the same time. Furthermore, if suspended rules exist, all of them are suspended due to the same arguments.
BABEL programs can be transformed to semantically equivalent non-subunifiable programs. For instance, Π_1 and Π_2 below are two different ways to transform the example program Π_0 shown above to a non-subunifiable one:

$$\begin{array}{llll} \Pi_1: & R_1: & f\ 0\ \ 0 & := 0. \\ & R_2: & f\ (s\ X)\ 0 & := 1. \\ & R_{31}: & f\ 0\ \ (s(s\ Y)) & := 2. \\ & R_{32}: & f\ (s\ X)\ (s(s\ Y)) & := 2. \end{array} \qquad \begin{array}{llll} \Pi_2: & R_{1,2}: & f\ X\ 0 & := g\ X. \\ & R_3: & f\ X\ (s(s\ Y)) & := 2. \\ & R_4: & g\ 0 & := 0. \\ & R_5: & g\ (s\ X) & := 1. \end{array}$$

For Π_1 the expression $(f\ (f\ X\ Y)\ Z)$ has now only one reducible subexpression, namely $(f\ X\ Y)$, and all the rules are suspended due to the first argument. For Π_2 the subexpression $(f\ X\ Y)$ is not a lazy(!) redex. The only possibilities are to apply rule $R_{1,2}$ or R_3 to the whole expression.

3.2 Uniform BABEL Programs

Unfortunately, non-subunifiable programs do not solve all the problems. Different rules may require the evaluation of arguments at different depths (for example rules $R_{1,2}$ and R_3 of Π_2). The applicability of the rules can be checked most efficiently in a situation where only the topmost constructor of each demanded argument has to be inspected. Moreover, it would then be sufficient to evaluate the demanded arguments to *head normal form* (i.e. *variable* or *construction*), and to do this once for all the rules. To get this situation, the program has to be transformed to a *flat* form, i.e. a form, where every term on the lhs of a rule is either a variable or a constructor applied to some (possibly 0) variables.
Now, we are ready to define the subclass of BABEL programs, in which we are interested:

> A *uniform BABEL program* is a BABEL program, which has only flat rules with pairwise non-subunifiable left hand sides. For such a program we say that the jth argument of a defined function f is *demanded* if the jth arguments of some (and hence all) program rules for f are non-variable terms.

We now show algorithms which transform any given BABEL program first to flat and then to uniform form. The result is a *quasi* BABEL *program* that may syntactically (but not semantically) violate the nonambiguity restrictions. Of course, this causes no problem for the implementation.

Step 1: Flatness

Algorithm FT: Input: A BABEL program Π.
Output: A flat (quasi) BABEL program $FT(\Pi)$.

$FT(\Pi) :=$ <u>if</u> all rules in Π are flat <u>then</u> Π
<u>else</u> <u>let</u> $f\ t_1\ \ldots\ t_n\ :=\ M$ in Π be such that (for some i, j, m: $1 \leq i \leq n,\ \ 1 \leq j \leq m$)
$t_i\ = (c\ s_1 \ldots s_m)$ and s_j is not a variable
<u>in</u> $FT(\Pi')$
<u>where</u>
$\Pi' = (\Pi\ -\ \{f\ t_1\ \ldots\ t_n\ :=\ M\}) \cup$
$\{f\ t_1 \ldots t_{i-1}\ (c\ X_1 \ldots X_m) t_{i+1} \ldots t_n\ := f'\ t_1 \ldots t_{i-1} X_1 \ldots X_m t_{i+1} \ldots t_n\} \cup$
$\{f'\ t_1\ \ldots\ t_{i-1}\ s_1\ \ldots\ s_m\ t_{i+1}\ \ldots\ t_n\ :=\ M\}$
/* $X_1, \ldots, X_m$ are new variables not appearing in $t_1, \ldots, t_n$ and f' is a new function symbol not appearing in Π. */

It is easy to prove that FT always finishes, $FT(\Pi)$ is a flat quasi program, and Π and $FT(\Pi)$ are semantically equivalent with respect to expressions using symbols of Π's alphabet. Applying algorithm FT to Π_0 produces the following result:

Π_0' :	R_1, R_2	as before
	R_3' :	$f\ X\ (s\ Y) := g\ X\ Y.$
	R_6 :	$g\ X\ (s\ Y) := 2.$

Some optimizations, which might be applied to the result produced by FT, will be discussed later.

Step 2: Uniformity

Algorithm UT: Input: A flat (quasi) BABEL program Π (output of FT).
Output: A uniform (quasi) BABEL program $UT(\Pi)$.

$UT(\Pi) :=$ <u>if</u> there is no pair of subunifiable rules in Π <u>then</u> Π
<u>else</u> <u>let</u> $f\ t_1\ \ldots\ t_n\ :=\ M_1,\ \ f\ s_1\ \ldots\ s_n\ :=\ M_2$ be two subunifiable rules in Π
where $t_i = X$ and $s_i\ =\ (c\ Y_1 \ldots Y_m)$ (for some $i, m:\ \ 1 \leq i \leq n,\ m \geq 0$)
in $UT(\Pi')$

where

$\Pi' = (\Pi \ - \ \{f \ s_1 \ \dots \ s_n \ := \ M_2\}) \cup$
$\{f \ s_1 \dots s_{i-1} \ Y \ s_{i+1} \dots s_n \ := f' \ s_1 \dots s_{i-1} \ Y \ s_{i+1} \dots s_n\} \cup$
$\{f' \ s_1 \ \dots \ s_n \ := \ M_2\}$

/* f' is a new function symbol not appearing in Π with the same arity as f and Y is a new variable not appearing in $s_1, \dots, s_n, M_2$. */

It is possible to prove that UT always finishes. $UT(\Pi)$ is an uniform (quasi) BABEL program and semantically equivalent to Π. Both algorithms can be optimized in a straightforward way avoiding the repetition of non-variable terms on both sides of a new rule and treating all the non-flat terms in one step. Also, rules with left hand sides which are identical up to the naming of variables can be joined in certain cases. For the presentation of the algorithms, we have attached importance to the readability instead of including all optimizations. Algorithm UT can be straightforwardly extended to general (non-flat) programs.
Applying algorithm UT to the program Π'_0 leads to the program Π'_1. If we use the mentioned optimizations, we obtain the program Π''_1.

$$\Pi'_1 : \quad \begin{array}{ll} f \ X \ \ 0 & := f' \ X \ 0. \\ f \ X \ \ 0 & := f'' \ X \ 0. \\ f \ X \ (s \ Y) & := g \ X \ Y. \\ g \ X \ (s \ Y) & := 2. \\ f' \ 0 \ \ 0 & := 0. \\ f'' \ (s \ X) \ 0 & := 1. \end{array} \qquad \Pi''_1 : \quad \begin{array}{ll} f \ X \ \ 0 & := f' \ X. \\ f \ X \ (s \ Y) & := g \ Y. \\ g \ (s \ Y) & := 2. \\ f' \ 0 & := 0. \\ f' \ (s \ X) & := 1. \end{array}$$

To close this section, let us make two remarks.

- Our algorithm to generate non-subunifiable programs differs from the algorithm given in [Echahed 88]. Echahed's solution replaces a rule with a variable which causes subunification by a set of rules, where this variable is replaced by terms of different patterns, as in program Π_1 above. Unfortunately, this transformation does not preserve the semantics if partial functions and infinite terms are allowed. Moreover, the outcomes computed for goal expressions are not preserved, in general. For instance, the only outcome for the expression $(f \ X \ (s(s \ 0)))$ with the rules of Π_0 is $(2,\epsilon)$ (result 2 and no binding for X). Echahed's transformation gives program Π_1 which leads to the outcomes $(2, \{X/0\})$ and $(2, \{X/s(Y)\})$. Our transformation preserves outcomes. [de Frutos, Fernández 90] have applied our transformation to investigate complete narrowing strategies for a subset of BABEL.

- The transformation of general BABEL programs in uniform programs introduces no significant inefficiencies. The size of the transformed program is linear in the size of the original one (cfr. [de Frutos, Fernández 90]). Moreover, if the optimized transformation is used, each rule application in the original program corresponds to a statically bounded number of rule applications in the transformed one, where no new unification steps (except with variables) are performed. For instance, the evaluation of $(f \ 0 \ (s \ 0))$ fails immediately using Π_0, while this failure is detected after one step by the rule for g in Π''_1.

4 The Lazy BABEL Machine

The lazy BABEL machine (*LBAM*) implements a lazy evaluation strategy for uniform programs. Argument expressions are narrowed to head normal form only if this is demanded by the defining rules of the applied function. The uniformity of programs allows to follow this strategy, because:

- we know which arguments demand evaluation, and they do it for every rule (*non-subunifiability*);

furthermore

- Constructor Nodes:

CONSTR	constructor name	pointers at components

- Variable Nodes:
  - Unbound Variable Nodes: UBV
  - Bound Variable Nodes:

VAR	graph-address

- Task Nodes:

TASK	code address	argument list	evaluation mode fields	continuation label	father address	status flag
status information						

Figure 1: Structure of Graph Nodes

- the demanded arguments only need evaluation to head normal form (*flatness*).

Note that the evaluation of demanded arguments to head normal form can be performed before trying the application of the defining rules. This has the advantage that all program rules are tried before backtracking is activated to reevaluate arguments. This may avoid nontermination in cases where infinitely many narrowings of one and the same argument are possible.
Program rules are always tried in their textual ordering. Backtracking is activated whenever a failure or a user's request for alternative solutions occurs. Primitive symbols are handled as if the user had introduced them through their predefined rules.
The machine consists of three components:

- the *program store* that contains the translation of a uniform BABEL program to machine code,
- the *graph* which contains *constructor, variable* and *task nodes*,
- the *active task pointer* which points at the task node that represents the currently executed function call.

Evaluation is controlled by the *task nodes* within the graph which correspond to activation records of function calls, but contain more information than in the case of purely functional languages. Among others, task nodes contain a *local stack* for graph manipulations, a *local program counter* and a *local trail* used to keep track of variable bindings that must be undone on backtracking.

4.1 Structure of the Graph Component

The graph is modelled as a mapping from graph addresses to graph nodes. Figure 1 indicates the structure of the different graph nodes. *Constructor nodes* are used to build structured data. *Variable nodes* represent logical variables. They are needed for the organization of unification. We distinguish *unbound* and *bound variable nodes.*
Task nodes represent applications, where the function is given by the address of the first line of its code and the arguments are given by pointers at their graph representations. Two *evaluation mode* fields indicate whether evaluation to head normal form (*hnf*) or to normal form (*nf*) is needed and whether the task node corresponds to an argument (*argev*) or body evaluation (*bdev*) – the latter causes a slightly different behaviour when the task terminates. The *continuation label* is the program address where the *activator task* – i.e. the task that started evaluation of the present subtask – has

local variables	program counter	local stack	return pointer
backtracking information			

Figure 2: Status Information for active and evaluated tasks

to continue if the present task terminates successfully. This label is saved by the subtask during the evaluation of its demanded arguments, which is done under the control of the activator. The *father address* of a task points at the task node of which it is an argument. In fact, a task node can be an argument of several nodes in the graph because of sharing. In this case, we choose among them the task node which demands its evaluation.

The *status* field of a task registrates its status, which may be *dormant*, *active* or *evaluated* depending on whether the evaluation of the task has not yet started, already started or successfully terminated. Depending on the status of a task, additional *status information* is provided within the task node.

The status information for *dormant* task nodes consists of a single field indicating the maximal number of local variables in the program rules for the task's function.

Active and *evaluated* task nodes contain the following status information, also displayed in Figure 2:

- the *local variables*, represented by pointers at variable nodes in the graph
- the *program counter*, which points at the currently executed instruction
- the *local stack*, consisting of graph addresses and used for graph manipulations
- the *return pointer*, pointing at the task to which control has to be returned on successful termination (the activator task). The activator and the father tasks (and, hence the return pointer and the father address) are not necessarily the same, because some argument may need no evaluation at the time when the function is called, but later.
- *Backtracking information*, explained below.

For the organization of backtracking, the status information of *active* and *evaluated* nodes must contain the following backtracking information (cfr. Figure 3), whose role is similar to that of choice points in Warren's abstract machine (WAM) [Warren 83].

local trail	backtracking pointer	last descendant pointer	backtracking address	program counter of the activator

Figure 3: Backtracking information

- The *local trail* is a list of graph addresses indicating bound variable nodes which must be overwritten by unbound variable nodes on backtracking.
- The *backtracking pointer* indicates the task that must be reactivated and forced to produce another result when the current task finally fails, i.e. no other function rule is applicable.
- The *last descendant pointer* is used to set the backtracking pointer of the next descendant of the task, i.e. the next task that is activated by the current task. When execution of a task starts, its last descendant pointer points at itself. After the successful termination of direct subtasks, this pointer indicates the last subtask that performed unifications (i.e. the task that most recently allowed an alternative computation). This is achieved by passing the last descendant pointer of a subtask to the activator task on termination.

The backtracking pointer of a newly activated task is initialized with the last descendant pointer of the activator task. This guarantees that in case of a failure the most recent reevaluable task is reactivated.

- The *backtracking address* is the program address of the next alternative rule that has to be tried on backtracking.
- Finally a copy of the *activator's program counter* (when evaluation of the current task starts) is saved in order to reset it on backtracking.

4.2 Behaviour of the Machine

Evaluation starts with an active task node that controls the execution of the program code for the goal. This code has the same structure as the code generated for the bodies of the program rules. Code generation will be explained in subsection 4.4 below.

The goal always needs full evaluation to *normal form*, while the bodies of function rules are evaluated to *head normal form (hnf)* or *normal form (nf)* depending on the evaluation mode of the function application that led to their evaluation. Evaluation modes are propagated dynamically. For simplicity, the *standard evaluation mode* of the machine is evaluation to *hnf*. As tasks may fail and partially generated results probably have to be resumed, it is in general not possible to force complete evaluation e.g. by a printing function as it is done in functional programming.

Conceptually, one way to guarantee evaluation to *nf* is to pass the expression as an argument to a strict identity function *nfe*, which can be programmed in BABEL as follows (due to the semantics of weak equality):

$$nfe\ X := (X = X) \rightarrow X.$$

Of course, a practical implementation should incorporate a more efficient mechanism for *nf* evaluation. The evaluation of a general expression starts with the generation of a graph representation for the expression using dormant task nodes for the representation of applications. Then, evaluation of the expression graph is initiated. Before a dormant task node –which will represent an application of some function f – becomes active, the arguments demanded by f's defining rules are evaluated to *hnf* under the control of the activator task. A dormant task node is activated (i.e. its status and backtracking information are initialized) when the evaluation of its demanded arguments to *hnf* has terminated and execution of the code for the application of the function rules is started. The function rules are treated one after the other, performing the unifications with the left hand sides to find an applicable rule and then evaluating the body of this rule. If this evaluation terminates successfully, the control is given back to the activator task. Whenever no rule is applicable, backtracking occurs and the last task that allowed an alternative computation is forced to backtrack. This task is indicated by the backtracking pointer.

Figure 4 indicates the graph structure that will be produced during the evaluation of the following simple example:

program rules:	$f\ X := h\ a\ (g\ X).$	$h\ a\ d := c.$	$g\ a := c.$ $g\ b := d.$

solve: $f\ X.$

4.3 Machine Instructions

Five classes of machine instructions are distinguished:

Stack instructions

- SKIP eliminates the top element of the local stack of the currently active task.

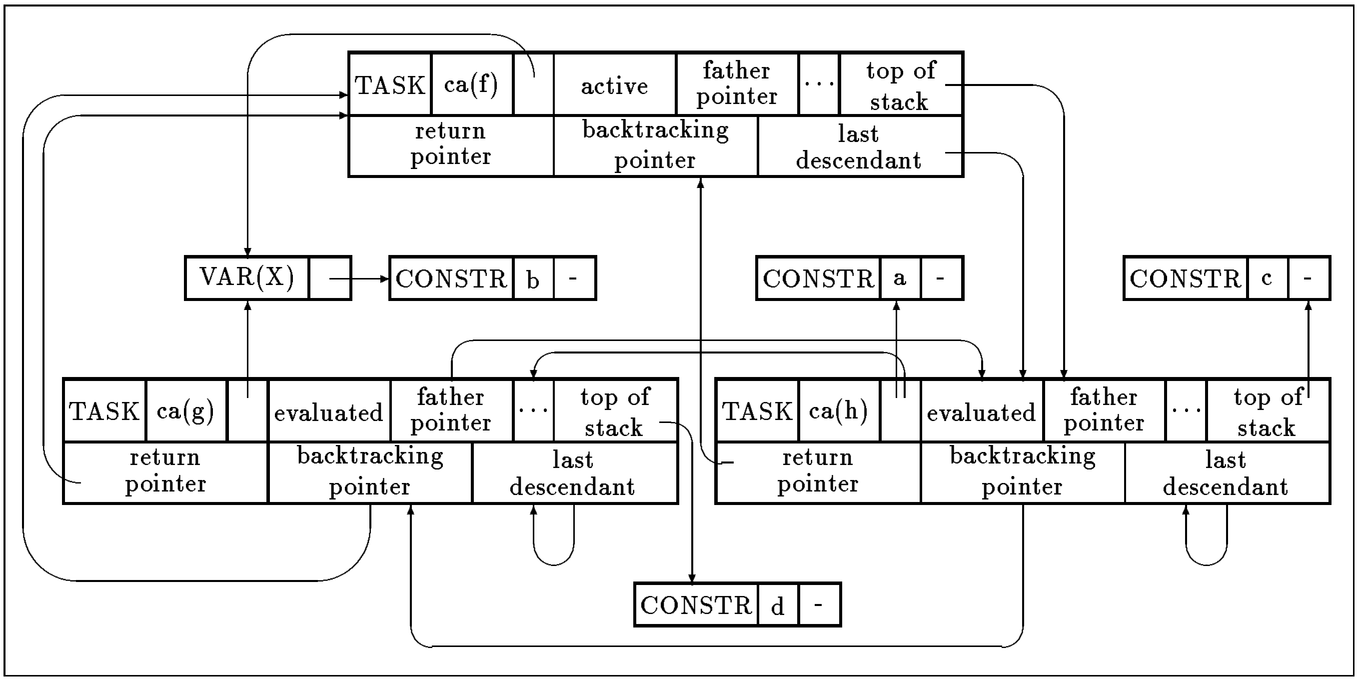

Figure 4: Example Graph

Graph instructions

Three kinds of LOAD-instructions allow to load graph addresses onto the top of the local stack of the currently active task:

- LOAD i loads the ith argument of the active task.
- LOADX i loads the ith local variable of the active task.
- LOADS i loads the ith argument of the dormant task node whose graph address is on top of the stack of the active task.

Furthermore, three instructions for the generation of new graph nodes are available:

- CONSTRNODE (c, n) generates a constructor node with constructor c and n graph addresses for the components of the structure taken from the local stack. The address of the newly generated node is pushed onto the local stack.
- NODE (ca, n, k) generates a dormant task node, where ca gives the code address of the function that has to be applied, n is the number of argument addresses given on the local stack and k gives the number of local variables occurring in the program rules of the function. The argument addresses on the stack will be replaced by the address of the newly generated task node.
- OBJNODE (ca, k) is a special instruction for the creation of the task node for the goal that must be distinguished to display the value of its variables together with the result of the computation. k gives the number of (local) variables occurring in the goal.

Unification instructions

- UNIFYCONST $(c, label)$ tries to unify the top element of the local stack with the constructor c. The top element of the stack must point at a constructor node or at a variable node.

  In the case of a constructor node with constructor c the argument list of this node is copied onto the stack. Further unification steps will be controlled by the code that follows the

UNIFYCONST-instruction. If the constructor name in the node is different from c backtracking occurs.

If the top of the stack points at an unbound variable node, this variable must be bound to a term whose top level constructor will be c. In this case a graph representation for the term must be constructed. The address of the code that generates such a graph representation is given as the third parameter (*label*) of the UNIFYCONST-instruction. Thus in the case of an unbound variable the program counter of the active task is set to *label*. After the construction of the graph the binding of the unbound variable to the new graph is performed by executing the instruction BIND.

- BIND expects a pointer at an unbound variable node and a pointer at a constructor node on top of the stack and binds the variable to the term graph whose root is the constructor node.
- UNIFYVAR i binds the ith local variable, that will be unbound when executing this instruction due to the linearity restriction of the BABEL-rules, to the expression whose graph address is on top of the local stack. The implementation of an occur-check and a general unification algorithm is not necessary at this place due to the left linearity.
- UNDO is used to delete variable bindings in case of backtracking.

Jump Instructions

- JMP *label* sets the program counter of the active task to *label*.
- JMF *label* causes a jump only if the boolean value *false* is represented by the top element of the local stack.
- JHNF *label* jumps to *label* if the pointer on top of the local stack represents an expression in head normal form, where bound variable nodes are dereferenced.
- JEM *label* performs a jump to *label* if the evaluation mode of the active task is *hnf*.

Process Instructions

The activation and termination of tasks is controlled by the following instructions:

- ARGEVALUATE and BODYEVALUATE perform a subroutine call to the dormant task whose address is given on top of the stack – called the *subtask* in the following. ARGEVALUATE is used to evaluate the subtask to *hnf* while BODYEVALUATE takes over the evaluation mode of the current task. In spite of the subroutine call, the current task – i.e. the *activator* – remains active in order to control the evaluation of the subtask's demanded arguments. For this purpose, the program counter of the activator is set to the address of the subtask's code, which starts with instructions for evaluation of the demanded arguments. The continuation label, i.e. the address of the instruction that follows the EVALUATE-instruction in the code of the activator, is saved by the subtask. Moreover, each argument of the subtask stores the subtask's address as father pointer in its own task node. This must be done because it is the only way to identify the father when a task has been reactivated by backtracking.
- EXECUTE transforms a dormant task into an active task by creating appropriate status and backtracking information. The program counter of the previously active task (activator task) is reset to the continuation label saved by the dormant task.
- RET is executed when a task terminates successfully and the control can be given back to the activator task. The status of the terminating task becomes *evaluated*. Its result remains on top of its local stack due to the possibility of sharing.

For the organization of backtracking the following instructions are necessary:

- BACKTRACK *label* sets the backtracking address of the active task to *label*.
- FAILRET will be executed when a task fails, i.e. no solution can be produced. The 'predecessor' of the task indicated by its backtracking pointer must then be reactivated and forced to evaluate in a different way.

To control the top-level behaviour of the machine, some more instructions are considered:

- MORE asks the user if more solutions are to be searched.
- FORCE forces the last successfully terminated task to backtrack and thus to compute more solutions.
- PRINTFAILURE finishes the whole execution with the output 'no (more) solutions'.
- PRINTRESULT is used to output a solution, which consists of the result value and bindings of the local variables within the goal.

4.4 Compilation of Uniform BABEL Programs

A uniform BABEL program consists of a set of flat, pairwise non-subunifiable rules and an expression (called the objective or goal), which is to be evaluated using the rules. The rules are grouped according to the function symbol they define. Hence, a uniform BABEL program can be represented as follows:

$$\begin{array}{l} \mathrm{PROC}(f_1, m_1, k_1, D(f_1)) \\ \ldots \\ \mathrm{PROC}(f_n, m_n, k_n, D(f_n)) \\ \mathrm{OBJECTIVE}(k_0) \end{array}$$

where $\mathrm{PROC}(f_i, m_i, k_i, D(f_i))$ denotes the set of rules defining function symbol f_i with arity m_i, k_i local variables and the set $D(f_i)$ of parameter indexes which are demanded by the rules for f_i $(1 \leq i \leq n)$. k_0 is the number of variables within the objective.

The code generation scheme for a BABEL program is shown in figure 5 (a). The code first generates a dormant task node for the objective, starts its evaluation and prints the result of the program after a successful evaluation. After this preliminary code, the translation of the procedures follows. This translation is done using the *proctrans* scheme. Finally, code for the objective is produced.

For a function symbol f with program-arity m, a set $D(f) = \{j_1, \ldots, j_k\}$ of demanded argument positions (t_{ij_l} is a non-variable term) and defining rules

$$\langle f\ t_{i1} \ldots t_{im} := M_i \rangle_{i=1}^{r}$$

the code that will be generated by the scheme *proctrans* is shown in figure 5 (b). The first part of the code for a function symbol performs the evaluation of those demanded arguments which are not yet in head normal form. This code will be executed under the direction of the activator of the task. After the evaluation of an argument, the pointer at the argument is skipped from the local stack of the activator. The result of the argument remains in the local stack of the argument's task node. When the evaluation of all the arguments is performed, the EXECUTE instruction activates the task and the defining rules of the function symbol are tested in their textual ordering. If all rules fail, the FAILRET command is used to force the predecessor of the task to backtrack.

The translation of each rule consists of code for the unification of the arguments of the function application with the terms on the left hand side of the rule and code for the evaluation of the body. Since the program is uniform and all demanded arguments have already been evaluated to head normal form, unification will either succeed or fail. For a rule $f\ t_1 \ldots t_m := M$ the code given in figure 5 (c) will be produced by the scheme *ruletrans*.

Furthermore, the following translation schemes are used to produce code for the unification and the evaluation of expressions:

(a) Code for a BABEL program:

0: OBJNODE (obj, k_0)
1: BODYEVALUATE
2: PRINTRESULT
3: MORE
4: JMF end
5: FORCE
proctrans ($\textsc{proc}(f_1, m_1, k_1, D(f_1))$)
$\vdots$
proctrans ($\textsc{proc}(f_n, m_n, k_n, D(f_n))$)
obj: EXECUTE
BACKTRACK fail
exptrans ($\textsc{objective}(k_0)$)
RET
fail: PRINTFAILURE
end: STOP.

(b) Code for a BABEL procedure:

$proctrans\ (\langle f\, t_{i1} \ldots t_{im} := M_i \rangle_{i=1}^{r}) :=$
$ca(f)$: LOADS j_1
JHNF l_1
ARGEVALUATE
l_1: SKIP
$\vdots$
LOADS j_k
JHNF l_k
ARGEVALUATE
l_k: SKIP
EXECUTE
BACKTRACK rule_2
ruletrans ($f\ t_{11} \ldots t_{1m} := M_1$)
rule_2: UNDO
BACKTRACK rule_3
ruletrans ($f\ t_{21} \ldots t_{2m} := M_2$)
rule_3: UNDO
$\vdots$
rule_r: UNDO
BACKTRACK l_{fail}
ruletrans ($f\ t_{r1} \ldots t_{rm} := M_r$)
l_{fail}: UNDO
FAILRET

(c) Code for a BABEL rule:

$ruletrans\ (f\, t_1 \ldots t_m := M) :=$
LOAD 1
unifytrans (t_1)
$\vdots$
LOAD m
unifytrans (t_m)
exptrans (M)
RET

Figure 5: Code Generation Schemes

- *unifytrans* : *Term* $\rightarrow$ *Code* generates code, which unifies a given term on the lhs of a rule with the corresponding argument given on top of the local stack.

$unifytrans\ (X_i) :=$ UNIFYVAR i
$unifytrans\ (c\ t_1 \ldots t_n) :=$ UNIFYCONSTR (c, l_{bind})
unifytrans (t_1)
$\vdots$
unifytrans (t_n)
JMP l_{end}
l_{bind} : *graphtrans* ($c\ t_1 \ldots t_n$)
BIND
l_{end} : ...

- *graphtrans* : *Exp* $\rightarrow$ *Code* produces code, which generates a graph representation of a given expression.

$graphtrans\ (X_i) :=$ LOADX i

$graphtrans\ (c\ M_1 \ldots M_n)$
$:=$ *graphtrans* (M_1)
$\vdots$
graphtrans (M_n)
CONSTRNODE (c, n)

$graphtrans\ (f\ M_1 \ldots M_n)$
$:=$ *graphtrans* (M_1)
$\vdots$
graphtrans (M_n)
NODE ($ca(f), n, k$)

- *exptrans* : *Exp* $\rightarrow$ *Code* produces code, which evaluates an expression by constructing its graph representation and evaluating this graph. If evaluation to normal form is necessary, a call to the function *nfe* is initiated.

$$
\begin{array}{rl}
\mathit{exptrans}\ (E) := & \mathit{graphtrans}\ (E) \\
 & \text{JEM } l_{\text{hnf}} \\
 & \text{NODE } (ca(\mathit{nfe}), 1, 1) \\
 & \text{BODYEVALUATE} \\
 & \text{JMP } l_{\text{end}} \\
l_{\text{hnf}}: & \text{JHNF } l_{\text{end}} \\
 & \text{BODYEVALUATE} \\
l_{\text{end}}: & \ldots
\end{array}
$$

The code for the three procedures in the small example program of subsection 2.3 is given in Figure 6. Additional code for primitive and arithmetic operations should be added. Instead of repeating the code sequence following the label l_{nf} three times we have introduced jump instructions JMP l_{nf}. Code generation for natural numbers has been simplified by using only one CONSTRNODE instruction, i.e. for example CONSTRNODE (2, 0) is an abbreviation of

CONSTRNODE (0, 0)
CONSTRNODE (suc, 1)
CONSTRNODE (suc, 1).

5 Extension to Higher Order BABEL

The extension of LBAM to cope with higher-order functions can be done similarly as in the innermost BABEL implementation in [Kuchen et al. 90]. Higher order BABEL does not allow higher order logical variables. Thus the unification algorithm does not need to be changed. The graph component of the machine is extended by so called *function nodes* that represent partial applications of functions defined by the program rules. Function nodes contain, in addition to the tag FUNCTION, the code address of the partially applied function, the partial argument list and the number of missing arguments:

FUNCTION	code address	partial argument list	number of missing arguments

Function nodes will be created by the instruction NODE, which is extended to compare the number of actually given and the number of needed arguments (provided as an additional parameter) for a user-defined function. The NODE instruction generates a dormant task if enough arguments are available and a function node otherwise.
A *new instruction* APPLY is introduced to extend function nodes by an additional argument and to create a dormant task node if the partial application is completed.

For *arbitrary applications* (MN), which cannot be compiled as applications of the form $(f\ M_1 \ldots M_n)$, the scheme *graphtrans* will generate the code

$$
\begin{array}{l}
\mathit{graphtrans}\ (M) \\
\mathit{graphtrans}\ (N) \\
\text{NODE } (ca(ap), 2, 0, 2).
\end{array}
$$

For the special *apply task* that is generated by the last NODE instruction the following code is provided:

0:	OBJNODE (obj, 1)
1:	BODYEVALUATE
2:	PRINTRESULT
3:	MORE
4:	JMF end
5:	FORCE
$ca(fib\_nbs)$:	EXECUTE
	BACKTRACK lb_1
	LOAD 1
	UNIFYVAR 1
	LOAD 2
	UNIFYVAR 2
	LOADX 1
	LOADX 2
	LOADX 1
	LOADX 2
	NODE ($ca(+)$,2,2)
	NODE ($ca(fib\_nbs)$,2,2)
	CONSTRNODE ($cons$, 2)
l_{nf} :	JEM l_{hnf}
	NODE ($ca(nfe)$,1,1)
	BODYEVALUATE
	RET
l_{hnf}:	JHNF l_{ret}
	BODYEVALUATE
l_{ret}:	RET
lb_1 :	UNDO
	FAILRET
$ca(nth)$:	LOADS 1
	JHNF l_1
	ARGEVALUATE
l_1:	SKIP
	LOADS 2
	JHNF l_2
	ARGEVALUATE
l_2:	SKIP
	EXECUTE
	BACKTRACK lb_2
	LOAD 1
	UNIFYCONSTR (0,bd_lb_1)
	JMP l_3
bd_lb_1:	CONSTRNODE (0,0)
	BIND
l_3:	LOAD 2
	UNIFYCONSTR ($cons$,bd_lb_2)
	UNIFYVAR 2
	UNIFYVAR 3
	JMP l_4
bd_lb_2:	LOADX 2
	LOADX 3
	CONSTRNODE ($cons$,2)
	BIND
l_4:	LOADX 2
	JMP l_{nf}
lb_2:	UNDO
	BACKTRACK lb_3
	LOAD 1
	UNIFYCONSTR (suc,bd_lb_3)
	UNIFYVAR 1
	JMP l_5
bd_lb_3:	LOADX 1
	CONSTRNODE (suc,1)
	BIND
l_5:	LOAD 2
	UNIFYCONSTR ($cons$,bd_lb_4)
	UNIFYVAR 2
	UNIFYVAR 3
	JMP l_6
bd_lb_4:	LOADX 2
	LOADX 3
	CONSTRNODE ($cons$,2)
	BIND
l_6:	LOADX 1
	LOADX 3
	NODE ($ca(nth)$,2,3)
	JMP l_{nf}
lb_3 :	UNDO
	FAILRET
$ca(fib\_nb)$:	EXECUTE
	BACKTRACK lb_4
	LOAD 1
	UNIFYVAR 1
	LOADX 2
	CONSTRNODE (1,0)
	CONSTRNODE (2,0)
	NODE ($ca(fib\_nbs)$,2,2)
	NODE ($ca(nth)$,2,3)
	LOADX 1
	NODE ($ca(=)$,2,2)
	CONSTRNODE ($true$,0)
	NODE ($ca(\rightarrow)$,2,1)
	JMP l_{nf}
lb_4 :	UNDO
	FAILRET
obj:	EXECUTE
	BACKTRACK $fail$
	LOADX 1
	CONSTRNODE (5,0)
	NODE ($ca(less)$,2,2)
	LOADX 1
	NODE ($ca(fib\_nb)$,1,2)
	NODE ($ca(,)$,2,1)
	JMP l_{nf}
$fail$:	PRINTFAILURE
end:	STOP

Figure 6: Code for the example program of subsection 2.3

ca(*ap*): LOADS 1
ARGEVALUATE
SKIP
EXECUTE
exptrans (*ap* X_1 X_2)
RET

The subscheme *graphtrans* of the scheme *exptrans* is extended by the following new case

graphtrans (*ap* X_1 X_2) := LOAD 1
LOAD 2
APPLY.

Arbitrary applications demand the evaluation of the first argument (functional expression) to head normal form, i.e. a partial application represented by a function node. Note that a higher order BABEL expression is in *head normal form* if it is a variable, a constructor application or a partial function application. The APPLY instruction is used to extend the partial application by the additional argument expression.

6 Prototype Implementation

A prototype implementation of the lazy BABEL machine has been programmed in PASCAL and used to test the machine's behaviour in simple examples. For this implementation, the LBAM has been extended by the predefined type *integer* and corresponding instructions (ADD, SUB, ...) to perform arithmetic operations. Moreover, a special treatment of constants (in order to generate nodes for them only once) and other optimizations have been included. We refer briefly to some of them in this section.

- A variable that appears as argument in the lhs of some rule is called a *temporal variable.* For instance, in the example of section 2.3 variables X and Y in the rule for *fib_nbs* are temporal. Temporal variables need no variable nodes; instead, the corresponding argument can be used directly.

- A task node is no longer needed as soon as no more alternatives are possible. This can already be detected when the execution of the last alternative finishes. A special LASTRET instruction can be used instead of the usual RET instruction in such a situation. A similar optimization is performed in the WAM [Warren 83], which uses either TryMeElse, RetryMeElse or TrustMeElseFail depending on the existence or nonexistence of alternatives. The task node is not needed for further computations, the result of the task remains in its local stack and may be useful later. These tasks are converted in so-called *semi-dormant tasks.* A detailed formal description of this optimization can be found in [Moreno et al. 90].

- It may be the case that a complicated graph representation is constructed for some argument expression whose evaluation is not demanded later. This will happen for functions that sometimes do not depend on some argument (for instance, sequential conjunction does not need to evaluate its second argument if the first one has been evaluated to 'false'). To avoid this source of inefficiency, the generation of graph representations for argument expressions should be delayed until they are really needed. One way to achieve this is to modify the machine by techniques similar to those in [Fairbairn,Wray 87]. Another possibility is to transform the BABEL program prior to compilation. For instance, an expression of the form (B_1, B_2) – where B_2 is some complicated expression using variables $X_1, \ldots, X_k$ – could be replaced by the transformed expression $(B_1, (aux\ X_1 \ldots X_k))$, where the following new defining rule should be added to the program:

$$aux\ X_1\ \ldots\ X_k\ :=\ B_2.$$

We plan to develop more efficient and elaborated implementations of the LBAM, including further optimizations, such as efficient implementations of BABEL's primitives, special treatment of lists, optimization of tail recursion, and avoiding of backtracking attempts in the case of purely functional computations.

7 Conclusions and Related Work

We have presented an approach to extend a functional graph reduction machine by unification and backtracking facilities, in order to achieve a compiled implementation of the first order core of the functional logic language BABEL, which integrates functional programmming and Horn clause logic programming by using lazy narrowing as operational semantics. We have shown that efficient compilation of lazy narrowing poses some significant additional difficulties in comparison with lazy reduction. As a solution, we have proposed an automatic transformation of programs into uniform programs, prior to compilation. This transformation does not introduce significant inefficiencies and has a number of advantages: Demanded arguments can be easily detected and evaluated to head normal form before trying to apply rules by unification. This ensures that all rules are tried for a fixed *hnf* evaluation of demanded arguments before backtracking for arguments' reevaluation is activated, which tends to avoid nontermination more often.
As sketched in section 5 our approach can be easily extended to full BABEL, which includes higher order functions and polymorphic types [Kuchen et al. 90]. Moreover, we expect that the decentralized structure of the LBAM will simplify its eventual evolution to a distributed parallel machine, based on an architecture similar to that of the functional parallel machine of [Loogen et al. 89].

The effort invested up to now in investigating implementation techniques for logic + functional languages, has been relatively small in comparison with the number of approaches and proposals for the integration [de Groot, Lindstrom 86], [Bellia, Levi 86]. Let us comment some approaches to the implementation of languages which are close to BABEL in some essential features.
[Lindstrom 87] extends a distributed graph reduction machine to allow logic variables, while preserving lazy evaluation and determinacy. The aim is to achieve an efficient implementation of AND-parallelism on distributed architectures. Backtracking is not supported.
An implementation of the logic + functional language K-LEAF [Giovannetti et al. 90] has been presented in [Balboni et al. 90], [Bosco et al. 90]. The language supports partial lazy functions, but is based on Horn clause logic with equality. Programs are flattened to Horn clauses and then compiled into code for an extension of the WAM [Warren 83] which supports a suspension/reactivation mechanism to treat lazy evaluation. The richer higher order logic + functional language IDEAL [Bosco, Giovannetti 86] is implemented on top of K-LEAF by translating IDEAL programs to K-LEAF programs. Our machine LBAM is not an extension of the WAM, but an extension of a graph reduction machine by WAM-like techniques. Moreover, we use flattening only for the left hand sides of rules during the transformation of programs to uniform programs, prior to compilation; right hand sides of BABEL rules are never flattened.
An interpreted implementation of narrowing has been presented in [Josephson, Dershowitz 89]. A compiled implementation of narrowing will appear in [Mück 90], but we have had no access to this paper yet. [Hanus 90] gives a compiled implementation of an Algebraic Logic Functional Language (ALF), based on an extension of the WAM. However, basic innermost narrowing is used, and hence lazy evaluation is not supported. The same limitation applies to the logic + functional language SLOG [Fribourg 85], which is based on equational Horn clauses and was implemented by an interpreter.

References

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

Oriented, Functional and Logic, Wiley 1989, 175–214.
[Bellia, Levi 86] M. Bellia, G. Levi: *The Relation between Logic and Functional Languages*, Journal of Logic Prog., Vol.3, 1986, 217–236.
[Bosco et al. 89] P. G. Bosco, C. Cecchi, C. Moiso: *An extension of WAM for K-LEAF: A WAM-based compilation of conditional narrowing*, Proc. Conf. on Logic Prog., MIT Press 1989, 318–333.
[Bosco, Giovannetti 86] P. G. Bosco, E. Giovannetti: *IDEAL: An IDeal DEductive Applicative Language*, Proc. IEEE Symp. on Logic Prog., IEEE Comp. Soc. Press 1986, 89–94.
[Echahed 88] R. Echahed: *On completeness of narrowing strategies*, CAAP 1988, LNCS 299, Springer 1988, 89–101.
[Fairbairn,Wray 87] J. Fairbairn, S. C. Wray: *TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators*, Conf. on Func. Prog. and Comp. Architec., LNCS 274, Springer 1987, 34–59.
[Fribourg 85] L. Fribourg: *SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting*, Proc. Symp. on Logic Prog., IEEE Comp. Soc. Press 1985, 172–184.
[de Frutos, Fernández 90] D. de Frutos-Escrig, M. I. Fernández-Camacho: *On Narrowing Strategies for Partial Non-Strict Functions*, 1990 (submitted for publication).
[Giovannetti et al. 90] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: *Kernel LEAF: A Logic plus Functional Language*, to appear in JCSS.
[DeGroot, Lindstrom 86] D. DeGroot, G. Lindstrom: *Logic Programming: Functions, Relations, Equations*, Prentice Hall, 1986.
[Hanus 90] M. Hanus: *Compiling Logic Programs with Equality*, to appear in Proc. Workshop on Prog. Language Implementation and Logic Prog., Linköping, Sweden, LNCS, Springer 1990.
[Josephson, Dershowitz 89] A. Josephson, N. Dershowitz: *An Implementation of Narrowing*, Journal of Logic Prog. 6, 1989, 55–77.
[Kuchen et al. 90] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, M. Rodríguez-Artalejo: *Graph-Based Implementation of a Functional Logic Language*, European Symp. on Prog. (ESOP) 1990, LNCS 432, Springer 1990, 271–290.
[Lindstrom 87] G. Lindstrom: *Implementing Logical Variables on a Graph Reduction Architecture*, Workshop on Graph Reduction, LNCS 279, Springer 1987, 328–400.
[Loogen et al. 89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: *Distributed Implementation of Programmed Graph Reduction*, Conf. on Parallel Architectures and Languages Europe (PARLE) 1989, LNCS 365, Springer 1989, 136–157.
[Moreno et al. 90] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: *Lazy Narrowing in a Graph Machine*, Aachener Informatik-Berichte Nr. 90–11, RWTH Aachen, 1990.
[Moreno, Rodríguez 88] J. J. Moreno-Navarro, M. Rodríguez-Artalejo: *BABEL: A functional and logic programming language based on constructor discipline and narrowing*, Conf. on Algebraic and Logic Prog., LNCS 343, Springer 1989, 223–232.
[Moreno, Rodríguez 89] J. J. Moreno-Navarro, M. Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Technical Report DIA/89/3, Universidad Complutense, Madrid 1989, to appear in the J. of Logic Prog.
[Moreno 89] J. J. Moreno-Navarro: *Diseño, semántica e implementación de BABEL, un lenguaje que integra la programación funcional y lógica*, Ph.D. Thesis, Facultad de Informática UPM, Madrid, July 1989, (in spanish).
[Mück 90] A. Mück: *Compilation of Narrowing*, to appear in Proc. Workshop on Prog. Language Implementation and Logic Prog., Linköping, Sweden, LNCS, Springer 1990.
[Reddy 85] U. S. Reddy: *Narrowing as the Operational Semantics of Functional Languages*, IEEE Int. Symp. on Logic Prog., IEEE Computer Society Press 1985, 138–151.
[Reddy 87] U. S. Reddy: *Functional Logic Languages, Part I*, Workshop on Graph Reduction, LNCS 279, Springer 1987, 401–425.
[Warren 83] D. H. D. Warren: *An Abstract PROLOG Instruction Set*, Technical Note 309, SRI International, Menlo Park, California, 1983.