Proc. ESOP 90, LNCS 432, Springer Verlag 1990, 271–290

Graph-based Implementation of a Functional Logic Language*

Herbert Kuchen, Rita Loogen
RWTH Aachen†

Juan José Moreno-Navarro
Universidad Politécnica de Madrid‡

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid§

Abstract

We describe in this paper a *graph narrowing machine* that has been designed for the implementation of a higher-order functional logic language. To execute functional logic programs the machine must be capable of performing *unification* and *backtracking*. Some details about the implementation of the new machine on an Occam/transputer system are given.

1 Introduction

During the last years, several attempts have been made to achieve an integration of functional and logic programming languages in order to combine the advantages of the different programming paradigms in a single framework [DeGroot, Lindstrom 86], [Bellia, Levi 86]. Usually one argues that logic languages have more expressive power than functional languages, while the latter have a simpler execution model, particularly suited to parallel implementations.

We investigate in this paper the implementation of the functional logic language BABEL [Moreno, Rodríguez 89] on a graph reduction architecture. We extend the original version of BABEL, which was first-order and type-free, to a higher-order functional logic language with a polymorphic type discipline. Its operational semantics is based on narrowing, an evaluation mechanism that uses unification for parameter passing [Reddy 85], [Reddy 87].

Our approach has been to extend a graph reduction machine[1] that has been designed for the execution of functional programming languages [Loogen et al. 89] by features that are necessary to execute functional logic programs, i.e. by unification and backtracking facilities. Accordingly, we may guarantee that purely functional programs can be executed almost as efficient as in the original machine. Of course, some overhead due to the different parameter passing mechanism cannot be avoided. But our way of implementing narrowing seems to be an interesting alternative to the reduction of narrowing to SLD-resolution, as it has been proposed by several researchers; cfr. e.g. [van Emden, Yukawa 87], [Bosco, Giovannetti, Moiso 88].

The BABEL abstract machine is controlled by machine code. We describe the compilation of BABEL programs into this abstract code. The compilation rules are very similar to the functional

*This work has been partially supported by a german-spanish cooperation action, funded by the german D.A.A.D. and the spanish M.E.C

†Lehrstuhl für Informatik II, Ahornstraße 55, 5100 Aachen, West Germany

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

§Departamento de Informática y Automática, Facultad de C.C. Matemáticas, 28040 Madrid, Spain

[1] more precisely the sequential kernel of a parallel graph reduction machine

case because backtracking will not be controlled by code but is performed by an implicit mechanism that is started by a special machine instruction.

For simplicity we discuss in this paper the realization of an innermost narrowing strategy, i.e. the arguments of a function call are evaluated before the function call.

This paper is organized as follows. In section 2 we give a short review of the language BABEL. The structure of the BABEL machine and the compilation of BABEL programs will be described in section 3. In section 4 we explain the organization of backtracking, i.e. the most important novel feature of the graph reduction machine. The BABEL machine has been implemented in OCCAM on a transputer. Some special aspects of this implementation are given in section 5. After the discussion of related work in section 6 we finally give some hints of future work. The transputer implementation will e.g. be used to incorporate the sequential BABEL machine in a parallel environment in order to exploit parallelism in the program execution whenever possible.

2 The functional logic language BABEL

The language BABEL has been designed by Mario Rodríguez-Artalejo and Juan José Moreno-Navarro to achieve integration of functional and logic programming in a flexible and mathematically well-founded way [Moreno, Rodríguez 88], [Moreno, Rodríguez 89]. It is based on a constructor discipline and uses narrowing as an evaluation mechanism; cfr. [Reddy 85], [Reddy 87]. In this paper we work with an extension of BABEL that supports polymorphic types and higher-order functions. However, higher-order *logic* variables are *not* allowed. Thus higher-order variables are never affected by narrowing.

Before we give the formal syntactic description of BABEL-programs, we consider a small example program:

```
typevar A, B.
data     list(A) = nil | cons (A, list(A)).

fun map: (A → B) → list(A) → list(B).
    map F []       := [].
    map F [X|Xs]   := [ F X | map F Xs ].

eval (map (+ 2) [X,3]) = [6,Y].
```

that will yield the boolean value true with variable bindings {X/4, Y/5}.

Note that we assume numbers and arithmetic operations as well as the equality operator to be predefined. We also allow a PROLOG-like syntax for lists.

The higher-order function map can be used in a very flexible way due to the first-order logic variables. The higher-order variable F can only be used as in functional programs.

Let $\Sigma = \langle TC_\Sigma, DC_\Sigma, FS_\Sigma \rangle$ be a polymorphic signature, i.e.

- TC_Σ is a set of *ranked type constructors* : tc/n, e.g. nat/0, list/1, ...
- DC_Σ is a set of *typed data constructors*: $c : \tau$, $c' : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$ with $\tau, \tau_i \in \mathbf{CType}_\Sigma$, where $\mathbf{CType}_\Sigma$ is the set of *constructed types* over type variables $\alpha, \beta, \gamma \ldots \in \mathbf{TVar}_\Sigma$ defined recursively by
$$\tau ::= \alpha \mid \mathrm{tc}/0 \mid \mathrm{tc}/n(\tau_1, \ldots, \tau_n) \quad \% \; n \geq 1.$$
- FS_Σ is the set of *typed function symbols* $f : \tau$ with $\tau \in \mathbf{Type}_\Sigma$, where $\mathbf{Type}_\Sigma$ is the set of *polymorphic types* defined by
$$\tau ::= \alpha \mid \mathrm{tc}/0 \mid \mathrm{tc}/n(\tau_1, \ldots, \tau_n) \mid (\tau \rightarrow \tau').$$

In the sequel we assume that "→" associates to the right and omit brackets accordingly.

We assume that Σ contains the predefined types:

- *bool* with data constructors true, false and function symbols

$$\begin{array}{lll} \neg & : & bool \to bool & \text{(logic negation)} \\ \wedge & : & bool \to bool \to bool & \text{(sequential and)} \\ \vee & : & bool \to bool \to bool & \text{(sequential or)} \end{array}$$

  and

- *nat* with data constructors 0 and s (successor function) and the usual arithmetic functions.

A special primitive function symbol is the strong equality symbol $=$ with type $\alpha \to \alpha \to bool$ whose definition will be given later.

In example programs we declare constructed types and data constructors in a MIRANDA-like style, cfr. [Turner 85].

We distinguish the following syntactic domains:

- *variables* ranged over by X, Y, Z ... $\in$ **Var**,

- *terms* ranged over by s, t, ... $\in \mathbf{Term}_\Sigma$:

$$\begin{array}{lll} t & ::= & X & \text{\% variable} \\ & | & c & \text{\% } c/0 \in DC_\Sigma\text{, constant} \\ & | & c(t_1, \ldots, t_n) & \text{\% } c/n \in DC_\Sigma\text{, construction,} \end{array}$$

- *expressions* ranged over by B,C,M,N ... $\in \mathbf{Exp}_\Sigma$

$$\begin{array}{llll} M & ::= & t & \text{\% term} \\ & | & c(M_1, \ldots, M_n) & \text{\% } c/n \in DC_\Sigma, \\ & | & f & \text{\% } f \in FS_\Sigma, \\ & | & (MN) & \text{\% application,} \\ & | & (B \to M) & \text{\% guarded expression, } B\text{: } bool \\ & | & (B \to M_1 \Box M_2) & \text{\% conditional expression, } B\text{: } bool, \\ & & & \text{\% } M_1, M_2 : \tau \text{ for some } \tau \end{array}$$

Expressions should be well-typed. We omit the formal definition of a type inference system for expressions; cfr. [Milner 78, Damas, Milner 82]. In the sequel we will reserve B, C for boolean expressions.

We remark that $B \to M$ and $B \to M_1 \Box M_2$ are intended to mean "**if** B **then** M **else** undefined" and "**if** B **then** M_1 **else** M_2," respectively.

We shall assume that application associates to the left and omit brackets accordingly.

A BABEL-*program* of signature Σ consists of a set of defining rules for the non predefined symbols in FS_Σ. The rules for the predefined symbols are implicitly added to every program, and will be presented later.

Notice that any $f \in FS_\Sigma$ must have type $\tau_1 \to \ldots \to \tau_n \to \tau$, for some $n \geq 0$ and some τ that is not of the form $\tau' \to \tau''$. Here, n is the *type-arity* of f.

Each defining rule for f must have the form

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

for some $m \leq n$ (called the *arity* of the rule) and satisfy the following restrictions:

1. *Flatness*: $t_i \in \mathbf{Term}_\Sigma$.
2. *Left Linearity*: $f\ t_1 \ldots t_n$ does not contain multiple variable occurrences.
3. *Well-Typedness*: Using appropriate type assumptions for the variables we may infer the types τ_i for the terms t_i $(1 \leq i \leq m)$, the type *bool* for the guard B and the type $\tau_{m+1} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ for the body M.
4. *Restrictions on free variables*: Any variable that occurs in the rhs but not in the lhs is called *free*. Occurrences of free variables are allowed in the guard, but not in the body. Moreover, free variables must be first-order. By this we mean that their types must be constructed types (under the type assumption used to well-type the rule).

In any program, all rules for a fixed f must have the same arity. This is called the *program-arity* of f, and is less or equal than f's type-arity. Programs are also required to satisfy a nonambiguity condition:

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

$$\begin{array}{lcl} f\ t_1 \ldots t_m & := & \{B \rightarrow\} M \\ f\ s_1 \ldots s_m & := & \{C \rightarrow\} N \end{array}$$

one of the three following cases must hold:

(a) *No superposition*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ are not unifiable.

(b) *Fusion of bodies*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ have a most general unifier (m.g.u.) σ such that $M\sigma, N\sigma$ are identical[2].

(c) *Incompatibility of guards*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ have a m.g.u. σ such that $(B \wedge C)\sigma$ is unsolvable.

This depends on a notion of unsolvability that must be chosen decidable and such that unsolvable boolean expressions cannot yield the value true under any valuation of their variables; cfr. [Moreno, Rodríguez 89].

We assume some predefined rules for the primitive function symbols and the guarded and conditional expressions. Among them, we have

- Rules for the boolean operations

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

- Rules for strong equality

$$\boxed{\begin{array}{ll} (c = c) := \text{true} & \%\ c/0 \in DC_\Sigma,\ \text{constant} \\ (c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n)) := (X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n) & \%\ c/n \in DC_\Sigma \\ (c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_m)) := \text{false} & \%\ c/n \in DC_\Sigma, \\ & \%\ d/m \in DC_\Sigma,\ \text{different} \end{array}}$$

- Rules for guarded and conditional expressions

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

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

The rules for $\wedge, \vee$ reflect the sequential character of these connectives. The rules for strong equality must be used respecting the types of constructors. They specify that an expression $(M_1 = M_2)$ will evaluate to true if M_1, M_2 evaluate both to the same term, and will evaluate to false if M_1, M_2 evaluate to different terms. $(M_1 = M_2)$ will be undefined if the evaluation of M_1 or M_2 does not terminate.

BABEL, as described in [Moreno, Rodríguez 89], supports infinite terms through lazy evaluation. This gives rise to a more sophisticated behaviour of strong equality.

As a last remark on BABEL programs, let us mention that pure PROLOG can be straightforwardly translated to BABEL. For instance, the PROLOG program

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs,Ys,Zs).
```

can be translated as follows:

```
typevar A.
data list(A) = nil | cons (A, list(A)).
fun  append: list(A) → list(A) → list(A) → bool.
     append [] Ys Zs := (Zs = Ys) → true.
     append [X|Xs] Ys [Z|Zs]
          := (Z=X ∧ (append Xs Ys Zs)) → true.
```

The idea is that PROLOG clauses translate into guarded BABEL rules whose body is identical to true. Strong equality must be used to ensure left linearity. Boolean valued functions play the role of predicates. In fact, a concrete BABEL implementation could allow a PROLOG-like syntax as a syntactical sugar, translating PROLOG-like programs to pure BABEL, cfr. [Moreno, Rodríguez 89].

Of course, append can also be programmed as a function in BABEL. The point is that the just given version behaves like the PROLOG append predicate under BABEL's evaluation mechanism.

A *goal* for a given BABEL Σ-program is any Σ-expression M which includes no higher-order variables. 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 will be called *outcome* in the sequel.

The restrictions imposed on higher-order variables are worth to be noted carefully. Higher-order variables may occur in the lhs of rules, but are forbidden to occur free in either rhs of rules or goals. This means that they are used only for rewriting, as in applicative functional programming.

The *narrowing semantics* of BABEL is based on the following *narrowing rule*:

> Let $ft_1 \dots t_m := R$ be a variant of a BABEL rule in the program which shares no variables with $(fM_1 \dots M_m)$. If there exists some most general unifier $\sigma \cup \theta$ (where σ binds variables in $(fM_1 \dots M_m)$ and θ binds variables in $(f\ t_1 \dots t_m)$) with
>
> $$t_i\theta = M_i\sigma \text{ for } 1 \leq i \leq m,$$
>
> then we may reduce
>
> $$(fM_1 \dots M_m) \longrightarrow_\sigma R\theta.$$

The *one-step narrowing relation*

$$M \Longrightarrow_\sigma N$$

where M, N are BABEL expressions and σ is a finite substitution of some first order variables occurring in M by terms, will be defined as follows

- $M_i \longrightarrow_\sigma N_i$ with $i \in \{1, \ldots, n\}$ implies
  - $c(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma c(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
  - $(M_1 \ldots M_i \ldots M_n) \Longrightarrow_\sigma (M_1\sigma \ldots N_i \ldots M_n\sigma)$
- $B \longrightarrow_\sigma B'$ implies
  - $(B \rightarrow M) \Longrightarrow_\sigma (B' \rightarrow M\sigma)$
  - $(B \rightarrow M_1 \Box M_2) \Longrightarrow_\sigma (B' \rightarrow M_1\sigma \Box M_2\sigma)$
- $M_i \longrightarrow_\sigma N_i$ with $i \in \{1, 2\}$ implies
  - $(B \rightarrow M_1) \Longrightarrow_\sigma (B\sigma \rightarrow N_1)$
  - $(B \rightarrow M_1 \Box M_2) \Longrightarrow_\sigma (B\sigma \rightarrow N_1 \Box M_2\sigma)$
  - $(B \rightarrow M_1 \Box M_2) \Longrightarrow_\sigma (B\sigma \rightarrow M_1\sigma \Box N_2)$

Narrowing of a BABEL-expression may have the following outcomes:

- *success:* $M \overset{*}{\Longrightarrow}_\sigma t$ with $t \in Term_\Sigma$
- *failure:* $M \overset{*}{\Longrightarrow}_\sigma N$, N is not further narrowable and $N \notin Term_\Sigma$
- *nontermination.*

For simplicity, we restrict ourselves in the following to an innermost narrowing strategy for the reduction of applications of user-defined functions. Of course, this implies that infinite objects cannot be used.

We now show the innermost evaluation of a goal for the map program. The redex at each step is underlined, and some intermediate steps (corresponding to predefined functions) are skipped.

$$
\begin{array}{ll}
 & \underline{\text{map } (+\ 2)[X,3]} = [6, Y] \\
\rightarrow & [\underline{(+\ 2\ X)} \mid \text{map } (+\ 2)[3]] = [6, Y] \\
\overset{*}{\rightarrow} & [s^2(X) \mid \underline{\text{map } (+\ 2)[3]}] = [6, Y] \\
\rightarrow & [s^2(X), \underline{(+\ 2\ 3)} \mid \text{map } (+\ 2)[\,]] = [6, Y] \\
\overset{*}{\rightarrow} & [s^2(X), 5 \mid \underline{\text{map } (+\ 2)[\,]}] = [6, Y] \\
\rightarrow & \underline{[s^2(X), 5] = [6, Y]} \\
\rightarrow & \underline{(s^2(X) = 6)} \wedge ([5] = [Y]) \\
\overset{*}{\rightarrow}_{\{X/4\}} & \underline{([5] = [Y])} \\
\rightarrow & (\underline{(5 = Y)} \wedge ([\,] = [\,])) \\
\overset{*}{\rightarrow}_{\{Y/5\}} & \text{true}
\end{array}
$$

The outcome of this derivation consists of result true and answer $\{X/4, Y/5\}$. Some predefined rules for $+$, $=$ and $\wedge$ have been used.

The BABEL machine tries the program's rules in their textual ordering and evaluates arguments from left to right; it backtracks when 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. However, this is not exactly so in all cases: Guarded and conditional expressions are handled in the usual non-strict way, i.e. evaluation of the guard before evaluation of the alternatives. The evaluation of the second member in conjunctions and disjunctions is avoided whenever possible, and strong equality is implemented through unification for the sake of efficiency. This implementation works fine with any equality $t_1 = t_2$ between two terms whose unification either succeeds or finitely fails; but it does not work properly in cases such as $X = c(Y)$, which allow for infinitely many outcomes with result false and different answers, according to the predefined rules of strong equality. The actual implementation will ignore these outcomes. The user can overcome this limitation by explicitly programming his own equality. The price to pay will be a risk of nontermination.

There is also a declarative — i.e. logical — semantics for first-order BABEL, related to a lazy version of narrowing through soundness and completeness results; cfr. [Moreno, Rodríguez 89].

3 Structure of the BABEL Machine

The BABEL machine is a sequential abstract graph narrowing machine by which the functional logic language BABEL will be implemented using an innermost evaluation strategy. This strategy has been chosen for simplicity in order to develop a first version of an abstract machine for BABEL.

The main component of the machine is a *graph*, which contains, among others, so called task nodes which correspond to ordinary activation records but contain much more information. A task node contains e.g. a local data stack for data manipulations and a local program counter. For the organization of backtracking a local trail is necessary. It is used in the same way as in the Warren Abstract Machine (WAM) [Warren 83] to keep track of variable bindings which must be removed in the case of backtracking.

Due to the use of local data stacks and local trails the machine has a very decentralized organization. This will simplify a later parallelization of the machine, i.e. the incorporation of the machine in a parallel environment.

The store of the BABEL machine consists of three components:

- the *program store* which contains the translations of the BABEL rules into machine code,
- the *graph*, which may contain task-, variable- and terminal nodes, and
- the *active task pointer* which points at the task node which corresponds to the currently executed procedure call. A BABEL procedure consists of all program rules corresponding to one function symbol.

Thus, we define

$$\mathcal{BAM} := \langle \underbrace{\mathrm{St}}_{\text{Store}} , \underbrace{\vdash}_{\substack{\text{Transition}\\ \text{relation}}} \rangle$$

where

$$\mathrm{St} := \mathrm{Program\_store} \times \mathrm{Active\_task\_pointer} \times \mathrm{Graph}.$$

A state of the machine is usually denoted by (p,atp,G), where p $\in$ Program_store, atp $\in$ Active_task_pointer and G $\in$ Graph.

3.1 The Graph Component

The graph component is modelled as a mapping from graph addresses into the graph nodes:

$$\begin{gathered}\mathrm{Graph} := \mathrm{Graph\_addresses} \rightarrow \mathrm{Graph\_nodes},\ \text{where}\\ \mathrm{Graph\_nodes} := \mathrm{Task\_nodes} \cup \mathrm{Terminal\_nodes} \cup \mathrm{Variable\_nodes}.\end{gathered}$$

Figure 1 indicates the structure of the different graph nodes. The computation is controlled by the *task nodes* which represent applications.

Each task node contains the address of the first line of code for the corresponding function symbol, pointers to the graph representation of the arguments and a list of pointers to unbound variable nodes which represent local variables. Depending on the status of a task — we distinguish *dormant*, *active* and *evaluated* tasks — additional information is provided within the task node. The

- Task Nodes:

TASK	argument list	status	local variables	code address
status-information (only if status = active or evaluated)				
backtracking information (only if status = active or evaluated)				

- Terminal Nodes:
  - Constructor Nodes:

CONSTR	constructor name	pointers to components

  - Function Nodes:

FUNCTION	partial argument list	code address
number of missing arguments	number of local variables	

- Variable Nodes:
  - Unbound Variable Nodes: | UBV |
  - Bound Variable Nodes: | VAR | Graph-address |

Figure 1: Structure of Graph Nodes

status-information of active and evaluated tasks consists of a local stack which is needed for the organization of data manipulations, a program counter which indicates the next instruction and a pointer to the father node, i.e. the node by which the current node has been created. This father pointer is used when a task finishes successfully and the control has to be returned to the father task.

Since BABEL is also a logic language, tasks must be able to perform backtracking. For this reason, active and evaluated task nodes contain certain *backtracking* information. This consists of a local trail which keeps track of variable bindings to be removed in case of backtracking; pointers to the 'previous brother' and the 'last son', which help to find the task to be reactivated when backtracking occurs; a backtracking address, which points to the code that must be executed if backtracking reactivates the task, and finally safe copies of the father's program counter, local stack and local trail, that are needed to restore the machine state in case of backtracking. All this will be explained in detail in section 4, where the organization of backtracking will be discussed.

In addition to the task nodes the graph contains *terminal* and *variable nodes.* Terminal nodes are constructor or function nodes.

Constructor nodes represent structured data. They contain the constructor name and a list of pointers to the graphs of the components of the structure.

Function nodes represent functional data which always correspond to partial function applications. Consequently a function node contains essentially the same base information as a task node (address of first line of code of the function, (partial) list of arguments, number of local variables) and additionally the number of arguments that are necessary to make the function application complete.

Variable nodes are needed for the organization of unification. We distinguish nodes for unbound and bound variables. Unbound variable nodes consist only of a tag (UBV). When a variable is bound to some terminal node the graph address of this node is written into the bound variable node which is indicated by the tag VAR. No more information needs to be stored for variables.

3.2 Machine Instructions

This subsection gives a short discussion of the machine instructions of $\mathcal{BAM}$. Five classes of machine instructions are distinguished.

3.2.1 Stack Instructions

Stack instructions are needed for an efficient implementation of some of the primitive functions of BABEL, especially for the logic negation (NOT) and the equality operator (CHECKEQ).

3.2.2 Graph Instructions

Graph instructions are used for the graph manipulation. By LOAD-instructions the addresses of arguments of the active task (LOAD i) or pointers to local variables (LOADX i) can be pushed onto the local stack, i.e. the stack within the task node of the active task.
Constructor nodes are generated by execution of the instruction

$$\text{CONSTRNODE } (c,\ m),$$

where c is the constructor name and m is the number of components whose addresses are taken from the local stack and replaced by the address of the new node.

Task and function nodes are created by the instruction

$$\text{NODE (ca, numarg, arity, locals)},$$

which has four parameters: the code address of the function, the number of arguments of the function application that are given on the stack, the program arity of the function, and finally the number of local variables. If enough arguments are given, i.e. numarg = arity, a dormant task node will be constructed. Otherwise, a function node will be created.

To add further arguments to a function node, we use the instruction APPLY i that expects i graph addresses and a pointer to a function node on top of the stack. The i graph addresses are added to the argument list of the function node. If this yields a complete application, a dormant task node is generated. Otherwise, a new function node is built. The existing function node must not be overwritten because of the possibility of sharing.

3.2.3 Unification Instructions

For the organization of unification we need the following instructions.

'UNIFYCONST (c, arity, label)' tries to unify the top element of the local stack with the constructor c. The top element of the data stack points at a constructor node or at a variable node.

In the case of a constructor node with constructor name 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, that expects a pointer to an unbound variable node and a pointer to a constructor node on top of the stack and that binds the variable to the term graph whose root is the constructor node.

The 'UNIFYVAR i'-instruction binds the i-th local variable, that will be unbound when executing this instruction due to the linearity restriction of the BABEL-rules, to the graph whose address is on top of the stack. The implementation of an occur-check and a general unification algorithm is not needed at this place due to the left linearity.

The UNDO-instruction is used to delete variable bindings in case of backtracking.

3.2.4 Control Instructions

Control instructions are jump instructions. The unconditional 'JMP label' sets the program counter of the active task to label. The conditional jump instructions 'JMT label' and 'JMF label' cause a jump only if the boolean value true and false, respectively, is represented by the top element of the stack.

3.2.5 Process Instructions

The activation and termination of tasks is controlled by the process instructions EVALUATE and RET, respectively. The EVALUATE-instruction performs a subroutine call to the dormant task whose address is given on top of the stack. The RET-instruction is executed when a task terminates successfully and the control can be given back to the father task.

For the organization of backtracking the instructions 'BACKTRACK label' and 'FAILRET' are necessary. The BACKTRACK-instruction initializes the backtracking information of a task. FAILRET will be executed when a task fails, i.e. no solution can be produced. The 'predecessor' of the task must then be reactivated and forced to evaluate in a different way. For more details, see the explanation of backtracking in section 4.

To control the behaviour of the machine on the top level, some more instructions are needed.

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

Before we discuss the organization of backtracking in detail, we show the translation of BABEL-programs into machine code by compiling the small map-example of section 2.

3.3 Compilation of BABEL-Programs

A BABEL program consists (mainly) of a set of 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 BABEL program looks like this:

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

where $\mathrm{PROC}(f_i, m_i, k_i)$ denotes the set of rules defining function symbol f_i with program-arity m_i and k_i local variables ($1 \leq i \leq n$). k_0 is the number of variables within the objective. The machine code for a BABEL program is the following:

map:	BACKTRACK nextrule	continue2:	LOADX 1
	LOAD 1		LOADX 2
	UNIFYVAR 1		APPLY 1
	LOAD 2		EVALUATE
	UNIFYCONST (nil, 0, bindlab)		LOADX 1
	JMP continue		LOADX 3
bindlab:	CONSTRNODE (nil, 0)		NODE (map, 2, 2, 3)
	BIND		EVALUATE
continue:	CONSTRNODE (nil, 0)		CONSTRNODE (cons, 2)
	RET		RET
nextrule:	UNDO	fail_lab:	UNDO
	BACKTRACK fail_lab		FAILRET
	LOAD 1	obj:	BACKTRACK last_fail
	UNIFYVAR 1		CONSTRNODE (1, 0)
	LOAD 2		NODE (+, 1, 2, 2)
	UNIFYCONST (cons, 2, bindlab2)		LOADX 1
	UNIFYVAR 2		CONSTRNODE (2, 0)
	UNIFYVAR 3		CONSTRNODE (cons, 2)
	JMP continue2		NODE (map, 2, 2, 3)
bindlab2:	LOADX 2		EVALUATE
	LOADX 3		RET
	CONSTRNODE (cons, 2)	last_fail:	PRINTFAILURE
	BIND		STOP

Figure 2: $\mathcal{BAM}$-**Code** for Example Program

0:	NODE (obj, 0, 0, k_0)
1:	EVALUATE
2:	PRINTRESULT
3:	MORE
4:	JMF end
5:	FORCE
	proctrans (PROC(f_1, m_1, k_1))
	...
	proctrans (PROC(f_n, m_n, k_n))
obj:	BACKTRACK last_fail
	exptrans (OBJECTIVE(k_0))
last_fail:	PRINTFAILURE
end:	STOP.

The first code generates a task node for the objective, starts evaluation by EVALUATE, and prints the result of the program after a successful evaluation. If the programmer asks for more solutions, the FORCE-instruction is executed and the task of the objective is forced to backtrack.

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. The BACKTRACK command stores the label to which to backtrack in case of a failure. The scheme *exptrans* produces code for the evaluation of the objective. If this evaluation fails finally this is reported by the PRINTFAILURE command.

For a function symbol f with program-arity m and defining rules

$$f t_{i1} \dots t_{im} = \text{body}_i \quad (1 \leq i \leq r)$$

the following code will be generated by the scheme *proctrans*:

$$
\begin{array}{rl}
 & \text{BACKTRACK label}_1 \\
 & \textit{ruletrans}\ (ft_{11}\ldots t_{1m} = \text{body}_1\ ) \\
\text{label}_1\text{:} & \text{UNDO} \\
 & \text{BACKTRACK label}_2 \\
 & \textit{ruletrans}\ (ft_{21}\ldots t_{2m} = \text{body}_2\ ) \\
\text{label}_2\text{:} & \text{UNDO} \\
 & \vdots \\
\text{label}_{r-1}\text{:} & \text{UNDO} \\
 & \text{BACKTRACK label}_r \\
 & \textit{ruletrans}\ (ft_{r1}\ldots t_{rm} = \text{body}_r\ ) \\
\text{label}_r\text{:} & \text{UNDO} \\
 & \text{FAILRET}
\end{array}
$$

The defining rules of a function symbol are tested in their textual ordering. If all rules fail, the FAILRET command is used to force the predecessor of a 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. For a rule

$$ft_1\ldots t_m = \text{body}$$

the following code will be produced by the scheme *ruletrans*:

$$
\begin{array}{l}
\text{LOAD 1} \\
\textit{unifytrans}\ (t_1) \\
\vdots \\
\text{LOAD m} \\
\textit{unifytrans}\ (t_m) \\
\textit{exptrans}\ (\text{body}) \\
\text{RET}
\end{array}
$$

The following translation schemes are used:

- *unifytrans* : $\mathbf{Term}_\Sigma \rightarrow \mathcal{BAM}$-**Code** generates code, which unifies an argument of the actual task with the corresponding term on the right hand side of a rule.

  unifytrans uses the scheme

- *graphtrans* : $\mathbf{Term}_\Sigma \rightarrow \mathcal{BAM}$-**Code** to produce code for the construction of a graph for a term, that has to be bound to an unbound variable in an argument of the actual task.

- *exptrans* : $\mathbf{Exp}_\Sigma \rightarrow \mathcal{BAM}$-**Code** produces code, which evaluates an expression to normal form (in particular the right hand side of a rule).

We will not go further into the details of the code generation. In figure 2 the translation of the map example (see section 2) is given.

The execution of a $\mathcal{BAM}$-program prog starts with the following initial configuration:

$$(\text{prog, atp}, G_0)$$

where G_0 is a graph which contains only one node representing the initial task (referenced by atp). This initial task is an active task node with the program counter initialized to 0 and its local stack and local trail are empty. The other informations are not needed.

local trail	backtracking pointer	last son pointer	backtrack address	safe copies of		
				program counter	local stack	local trail
				of the father		

Figure 3: Backtracking information

4 Backtracking

This section describes and justifies the backtracking information needed in task nodes. The content of the backtracking information that has been described in section 3.1 can be seen in figure 3. All this information is generated each time an EVALUATE command is executed. The local trail was described in section 3.1. Next, there are two pointers to task nodes in the graph: a so called *backtracking pointer* to the 'previous brother' (the node generated previously by the father) and a so called *last son pointer* to the 'last son' (the last son generated by the task).

The backtracking pointer is used when a task fails — there is no possibility of returning a value — to select the task that must then be re-evaluated. If a task is the first son of its father the backtracking pointer is the address of the father. Otherwise the backtracking pointer points at the task node that was activated by the father node just before this task.

The last son pointer is used to initialize the backtracking pointer of newly generated subtasks. It is also used to find the task to re-evaluate in case of backtracking. Of course, at any moment, the address of the active task is the last son pointer of its father.

Using these pointers one can determine an implicit stack of nodes that reflects the order in which nodes have been activated. The control of the BABEL machine is based on the following ideas:

- A task returning with success (RET command) gives the control to the father which continues its execution.
- A task returning with failure (FAILRET command) gives the control to the previous task in the implicit stack.

The recursive description of this stack in terms of the pointers in the graph is:

- The top of the stack is the right-most bottom-most element of the tree (following the last son pointer of all the nodes, beginning with the very first task generated).
- The predecessor of each task is:
    1. its father if its backtracking pointer points to the father. The task is the first son.
    2. Otherwise the predecessor node is the first node of the stack below the node indicated by its backtracking pointer. The predecessor can be determined by doing one step along the backtracking pointer and then following all the last son pointers.

The following example should clarify this description. Consider the following rules of a BABEL program

$$
\begin{array}{l}
f\ X := h\ a\ ((p\ X) \rightarrow (g\ X) \Box (l\ X)).\\
p\ X := (q\ X) \wedge (r\ X).\\
q\ a := \text{true}.\\
q\ b := \text{false}.\\
r\ a := \text{true}.\\
g\ X := k\ X.\\
l\ X := b.\\
k\ a := a.\\
h\ a\ b := a.
\end{array}
$$

and the objective $(f\ X)$.
The following picture indicates the graph structure with the different pointers:

Pointers:

Father Pointer,
Backtracking Pointer,
Last son pointer,
Implicit Stack

Tasks	T_0	T_1	T_2	T_3	T_4	T_5	T_6
Function symbols	f	p	q	r	g/l	k	h

Following the policy that a task is not removed from the stack until all its alternatives have been tried it is clear that no answer is lost.

But some other informations for handling backtracking are needed in task nodes. The *backtrack address* is the address of the program where the task must continue in case of backtracking (i.e. the address of the next rule). The rest of the backtracking information is a safe copy of the state of the father: a copy of the program counter, the stack and the local trail (in fact we only need a pointer to the top of the local trail in this moment). It is used to restore the state of the father in case of backtracking, undoing all the bad decisions. The copy is made during the execution of the EVALUATE command and the restoring is made along the searching path of the next node to re-evaluate in case of backtracking.

The computation of the previous example shows why these copies are needed. The call to function f creates a task (T_0) that is executed. In order to generate the code for the body of f the first argument of the call to h (constant a) is stored on the local stack. After this, a call to predicate p is done which generates calls to predicates q and r. The first value for p is true binding X to a.

The next step is to continue the execution of the $\mathcal{BAM}$ code for the body of f. The second argument of h is a conditional expression. The "then" expression is selected evaluating g (where evaluation of k is demanded) returning a to the local stack of T_0. Remember that another a was stored on this stack. These two values are eliminated from the stack to construct the task $(h\ a\ a)$. But this task fails (there is no rule for it).

The first task to be re-evaluated is T_5 (for k). In the case that the value of this function changes, we would need to restore the local stack of T_0 to preserve the arguments of h (the first a on the stack).

In our example the backtracking to task T_5 fails as well as the backtracking to T_4 (g). The condition $(p\ X)$ is re-evaluated. If the boolean value changed, the program counter of T_0 needs to be modified to execute again the code for selecting between the "then" or the "else" branch of the conditional. The backtracking to T_3 (r) also fails, and backtracking to T_2 is needed. For this purpose it is necessary to unbind the variable X because T_2 has bound it, i.e. to undo the binding of X noted in the trail of T_2.

5 Implementation

In the implementation of the BABEL Machine we have integrated some extensions and optimizations.

The extensions suppose the inclusion of new nodes: nodes for constants (created at the beginning of the execution and shared for all the nodes that use them) and nodes for numbers. The number nodes lead to new instructions (ADD, SUB, MUL, DIV, MOD) to execute operations, $+$, $-$, $*$, div, mod which can be used in BABEL programs as well as integer literals. The two most important optimizations are explained in the following.

5.1 Temporal variables

A *temporal variable* of a rule is a variable that occurs on the left hand side but not inside a construction (i.e. it is directly the argument of the function). Temporal variables do not need a variable node for themselves. The argument can be used directly instead of the variable. For instance, variable F in the map program is temporal. We can avoid the unification of variable 1 with argument 1 and use the instruction LOAD 1 instead of LOADX 1 in the translation of the body.

5.2 Disposition of task nodes

The second optimization deals with the *disposition of task nodes* and is related with the reusability of "binding frames" in the Warren abstract machine.

A task node is no longer needed as soon as the existence of no more alternatives to try is noticed (execution of a FAILRET). But this situation can be found out still earlier, when the task finishes the execution of the last alternative (RET command of the last rule). If the information stored in a task node is no longer useful the node can be disposed. By making a compromise between time and memory efficiency, one could dispose the node of a task that finishes the execution of the last alternative if

- the task has no sons, and
- its trail is empty or

  the task to be re-evaluated in case of failure is the father (its trail is appended to the trail of the father).

This optimization could be handled by a new instruction LASTRET that is used in the translation of the last rule of a function.

5.3 Garbage Collection

A *garbage collection* mechanism has been implemented too. Task nodes (and, hence, their variable nodes) are only disposed when one of the situations described before happens. For terminal nodes and number nodes a classical reference counter system is used.

5.4 Results

The current state of the implementation is the following. A Pascal version has been developped to test the behaviour of the machine. A more efficient implementation has been done in OCCAM to prepare the development of parallel distributed versions.

The early results show a good behaviour in time (similar to Prolog programs) and a little bit worse in memory. Hence, the optimizations described before are very important. More optimizations need to be included in future versions, for instance a detection of purely functional computations (generating smaller task nodes by avoiding the backtracking information), special treatment of tail recursion and specific instructions for lists.

6 Related Work

A different approach to the implementation of a higher-order functional logic language has been described in [Bosco et al. 89], [Balboni et al. 89],

7 Future Work

To cope with infinite objects and non-strict functions it is necessary to change the evaluation strategy of $\mathcal{BAM}$. The development of a lazy $\mathcal{BAM}$ which does not evaluate arguments unless they are needed, is in progress. This machine, on which we will report in a forthcoming paper, evaluates needed arguments of functions only up to head normal form. Sharing of graph nodes is more important for the lazy machine than for the innermost machine.

Another important research subject is the development of a parallel BABEL machine. In order to simplify the parallelization of $\mathcal{BAM}$, we have chosen a very decentralized structure for $\mathcal{BAM}$. Since we are interested in using large networks of processors, we prefer a loosely coupled architecture, which consists of several processors with local memory communicating by exchanging messages. Tightly coupled architectures are not considered since they are limited to a small number of processors.

The parallel $\mathcal{BAM}$ will have a structure similar to the parallel abstract machine PAM [Loogen et al. 89], which has been developped for the parallel implementation of functional languages on a loosely coupled network of processors.

Each processing unit of the PAM contains a communication unit and a reduction unit. The communication units are responsible for the exchange of messages, while each reduction unit represents a sequential graph reducer that has been extended for the integration in a parallel machine. In the parallel $\mathcal{BAM}$ each reduction unit is replaced by a narrowing unit, which is an analogous extension of the sequential $\mathcal{BAM}$.

The parallelization of the $\mathcal{BAM}$ is more complicated than the parallel implementation of a functional language. The reason is the occurrence of side effects caused by logic variables. Thus, more synchronization between parallel processes is needed. In the literature, AND- and OR-parallel implementations of logic programs are investigated. AND-parallelism can be seen as a parallel execution of the arguments of a function. In PROLOG this function is the AND-operation, in BABEL arbitrary functions can be used (like in functional languages). OR-parallelism uses a parallel execution of the different rules for each function symbol.

We are currently working on an AND- and on an OR-parallel version of the $\mathcal{BAM}$, that will be implemented on a transputer system (like the PAM).

References

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

[Bosco et al. 88] P.G. Bosco, E. Giovannetti, C. Moiso: *Narrowing versus SLD–resolution*, Theoretical Computer Science 59, 1988, pp. 3–23.

[Damas, Milner 82] L. Damas and R. Milner: *Principal type schemes for functional programs*, ACM Symp. on Principles of Programming Languages, 1982.

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

[van Emden, Yukawa 87] M.H.van Emden, K.Yukawa: *Logic Programming with Equations*, Journal of Logic Programming, Vol.4, 1987.

[Loogen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Distributed Implementation of Programmed Graph Reduction*, Proc. Conf. on Parallel Architectures and Languages Europe 1989, LNCS 365, Springer Verlag 1989.

[Milner 78] R. Milner: *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17(3), 1978.

[Moreno, Rodríguez 88] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *BABEL: A functional and logic programming language based on constructor discipline and narrowing*, In: I.Grabowski, P.Lescanne and W.Wechler (eds.), *Algebraic and Logic Programming*, LNCS 343, Springer Verlag, 1989.

[Moreno, Rodríguez 89] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming 1989, to appear.

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

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

[Turner 85] D.A.Turner: *Miranda: A non-strict functional language with polymorphic types*, Proc. ACM Conf. on Functional Languages and Computer Architecture 1985, LNCS 201, Springer Verlag 1985.

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