Independent AND-Parallel Narrowing

Herbert Kuchen — RWTH Aachen*

Juan José Moreno-Navarro, Manuel V. Hermenegildo — Universidad Politécnica de Madrid†

Abstract

We present a parallel graph narrowing machine, which is used to implement a functional logic language on a shared memory multiprocessor. It is an extension of an abstract machine for a purely functional language. The result is a programmed graph reduction machine which integrates the mechanisms of unification, backtracking, and independent and-parallelism. In the machine, the subexpressions of an expression can run in parallel. In the case of backtracking, the structure of an expression is used to avoid the reevaluation of subexpressions as far as possible. Deterministic computations are detected. Their results are maintained and need not be reevaluated after backtracking.

1 Introduction

During the past few years, several attempts have been made to achieve an integration of functional and logic programming languages, trying to combine the advantages of both paradigms. *Functional logic languages*, one class of such languages, have functional syntax and use narrowing, an evaluation mechanism that uses unification for parameter passing, as operational semantics [21]. Babel [19,20,11,12] is a functional logic language based on a constructor discipline defined in a simple, flexible, and mathematically well-founded way.

A natural way of implementing these languages is to extend functional programming techniques. A couple of graph narrowing machines have been defined for the sequential implementation of Babel. [11,12] follow an innermost strategy, while [18] uses lazy evaluation. All of them have been designed by extending a graph reduction machine for a functional language [15] with those features that are necessary for executing functional logic programs, i.e. unification, backtracking, and logical variables. The mechanisms used for both are inspired by the techniques used in the implementation of Prolog, mainly the ideas present in the Warren Abstract Machine (WAM) [24]. The approach contrasts with others, normally termed *Logic functional languages*, such as e.g. K-LEAF [1], where the converse solution of adding support for functional characteristics to a logic language is taken both at the language and implementation levels. One advantage of the functional logic language approach, is that it guarantees that purely functional programs can be executed almost as efficiently as in the original functional machine, although some overhead due to the different parameter passing mechanism cannot be completely avoided.

In this paper the subject of parallel execution in the context of the functional logic language Babel is treated. A particular form of parallelism is presented and an abstract machine for its implementation, which is essentially a parallel variant of the BAM [11], is then described. The parallelism exploited allows subexpressions of an expression to run in parallel, if they do not share a logical variable. This is essentially a generalization of the concept of independent and-parallelism, known from logic programming [6,3,10], where AND is the only "function" combining subexpressions. As in [15], we allow n expressions e_i ($1 \leq i \leq n$) to run in parallel with the expression e that uses it. The synchronization is, however, different from the usual one in independent and-parallel implementations of Prolog. In most such implementations, and due to the lack of information on directionality of arguments in logic programs (unless extensive global analysis is performed or the program is annotated), a computation e, which combines the results of some parallel computations e_i ($1 \leq i \leq n$), has to be placed and evaluated after them. In a language of the type being considered we can take advantage of functional dependencies so that the synchronization of e and each e_i is performed when e really needs the e_i. This increases the degree of parallelism because $n + 1$ tasks (instead of n, where the extra task could in turn generate more parallel tasks) run in parallel. The ability of the abstract machine of keeping multiple environments is used to take advantage of the structure of e to avoid reevaluations. When some e_i produces a new solution, only the work made after the use of this e_i needs to

*Lehrstuhl für Informatik II, Ahornstraße 55, 5100 Aachen, Germany, email: herbert@zeus.informatik.rwth-aachen.de

†Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain, email: {jjmoreno, herme}@fi.upm.es, fax: +341-352-4645

be redone, while for several reasons in Prolog implementations the evaluation of e generally restarts from the beginning.

Moreover, it is possible to keep some results of the parallel tasks, avoiding reevaluations. Since Babel is a functional language a Babel function maps each sequence of (ground) arguments to exactly one result. This property can be used to dynamically detect so-called *deterministic computations* [12,16]. Such a computation cannot produce an alternative result. If it is involved in a parallel execution, it is not affected by backtracking. Its result is kept and can be used later on, when the reexecution of the previous expressions is successful and the value is needed again.

This paper is organized as follows. In Section 2, we briefly introduce the language Babel and discuss extensions to Babel in order to be able to express parallelism. Section 3 presents the independent and-parallel Babel machine. Section 4 focuses on the backtracking mechanism of this machine. Section 5 presents some related work. Finally, in Section 6, we conclude and point out future work.

2 The Functional Logic Language Babel

The language Babel integrates functional and logic programming in a flexible and mathematically well-founded way. It is based on a constructor discipline and uses narrowing as evaluation mechanism [21]. Babel has a Miranda-like polymorphic type system [23]. Since herein we want to focus on the issue of parallel implementation, we will not consider higher order functions throughout this paper (although including them presents no additional problem). Also, rather than giving a precise description of the syntax and semantics of Babel (which can be found in [19,20,11,12]) we will present Babel in an intuitive way by first using a small example program for the well-known *towers* of Hanoi problem and then presenting some basic concepts:

datatype pin := a | b | c.
datatype nat := 1 | (suc nat).
datatype list α := nil | (cons α (list α)).
fun *towers*: pin $\rightarrow$ pin $\rightarrow$ pin $\rightarrow$ nat $\rightarrow$ (list (list pin)) $\rightarrow$ (list (list pin)).
    towers Source Help Dest 1 L := [[Source, Dest] | L].
    towers Source Help Dest (suc M) L :=
      towers Source Dest Help M [[Source,Dest] | (*towers* Help Source Dest M L)].

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

solve *towers* a b c N [ ].

which would return the list of moves for each value of N (number of disks) starting with 1, i.e. the first solution is the value [[a,c]] with binding {N/1}.

A *datatype* definition introduces a new ranked *type constructor* (e.g. pin, nat (both of arity 0), list (of arity 1)) and a sequence of *data constructors* (separated by |). If an n-ary type constructor ($n \geq 0$) is applied to n argument types (including *type variables* –like α– representing any valid type) the resulting type expression is a valid type, e.g. "list pin" and "list α". A consistent replacement of type variables in a *polymorphic type* (i.e. a type including type variables) by other types yields an *instance* of the polymorphic type.

A *data constructor* is an uninterpreted function mapping an instance of its argument type (given behind the name of the constructor) to the corresponding instance of the type on the left hand side of the datatype definition. For example, the most general type of "cons" is cons: $\alpha \rightarrow$ (list α) $\rightarrow$ (list α). But "cons" may also be applied to arguments which have a more specific type (i.e. an instance of the argument type), e.g. "cons 1 nil" is of type "list nat" (α is replaced by nat).

A Prolog-like syntax for lists is allowed, i.e. [e | e'] is equivalent to "cons e e'", [e, e'] represents "cons e (cons e' nil)" and [ ] is the empty list "nil".

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

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

where $t_1,\ldots,t_n$ are terms, b is a boolean expression, and e is an arbitrary expression. A *term* is either a (logical) *variable* (denoted by an identifier beginning with a capital letter) or an application of an n-ary data constructor ($n \geq 0$) to n argument terms. An *expression* has the form:

$$
\begin{array}{ll}
e ::= t & \%\ \text{term} \\
\quad \mid (c\ e_1 \ldots e_n) & \%\ \text{application of a } n\text{-ary data constructor } c\ (n \geq 0) \\
\quad \mid (f\ e_1 \ldots e_n) & \%\ \text{application of a } n\text{-ary function } f\ (n \geq 0) \\
\quad \mid (b \rightarrow e) & \%\ \text{guarded expression, meaning ``if } b \text{ then } e \text{ else undefined''} \\
\quad \mid (b \rightarrow e_1 \,\square\, e_2) & \%\ \text{conditional expression, meaning ``if } b \text{ then } e_1 \text{ else } e_2\text{''}
\end{array}
$$

All expressions have to be well typed. For an application this means that the types of the arguments have to be (consistent) instances of the argument type of the function. We assume that application associates to the left and omit parentheses accordingly.

Babel functions are functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is only one result. This is ensured by syntactic restrictions for the rules (see [12]). In order to simplify the translation of Babel, no variable may occur more than once on the left hand side (left linearity). The type bool is predefined in Babel: **datatype** bool := true | false. Moreover, several basic operations like equality (=), conjunction (,), disjunction (;) and so on are predefined. In [12], a straightforward translation of Prolog rules to Babel rules is shown.

2.1 Narrowing Semantics

The operational semantics of Babel is based on *narrowing*. To use narrowing as the operational semantics of (syntactically) functional languages was first proposed by Reddy [21]. An expression e is narrowed by applying the minimal substitution that makes it reducible, and then reducing it. The minimal substitution is found by unifying e with the left hand side of a rule. A new expression e' and an *answer substitution* σ binding some variables from e are the *outcome* of one narrowing step (denoted by $e \Rightarrow_\sigma e'$). If, after several steps, an expression e is narrowed to a term t, we speak of a *computation* for the *goal* e with *result* t and *answer* σ, where σ is the composition of the answer substitutions of all the individual steps. If e cannot be narrowed further, but it is not a term, then the computation *fails*. The machine will backtrack in such a situation. The rules are tried in their textual order.

The goal in the *towers* of Hanoi example can be narrowed as follows:

1. solution:	*towers* a b c N [ ]	$\Rightarrow_{\{N/1\}}$	[[a,c]]
2. solution:	*towers* a b c N [ ]	$\Rightarrow_{\{N/suc\ M\}}$	
	towers a c b M [[a,c] | *towers* b a c M [ ]]	$\Rightarrow_{\{M/1\}}$	
	towers a c b 1 [[a,c],[b,c]]	$\Rightarrow$	
	[[a,b],[a,c],[b,c]]		
3. solution:	...		

The outcome of the second solution consists of the result [[a,b],[a,c],[b,c]] and the answer {N/ suc 1}.

2.2 Extensions of Babel to Express Parallelism

An (intermediate) goal usually contains several subexpressions, which can in principle be narrowed in parallel. In the above example, the expression

towers a c b M [[a,c] | *towers* b a c M [ ]]

contains two applications of the function *towers*, namely e_1 := *towers* a c b M [...] and e_2 := *towers* b a c M [ ]. The resulting parallelism is a generalization of the and-parallelism of Prolog. While in Prolog the conjunction is the only way to connect applications of predicates, Babel allows arbitrarily nested function applications. Although the notion *subexpression parallelism* might be more appropriate, we will still call it and-parallelism.

2.2.1 Explicit Parallelism.

Clearly, it is not always advisable to narrow subexpressions in parallel. First of all, there is the issue of "granularity" (we will deal with other issues in the following section): for a "small" expression (in terms of execution time) it may be the case that spawning a process to evaluate it is more expensive than evaluating the expression sequentially. Although some work has been done on automatic granularity analysis [2,14] available results still need to be extended to functional logic languages and there are still few practical analysis tools. In this paper we adopt the solution of providing a special syntax which allows the expression of the desired parallelism. This allows the user, who is in the best position to determine which expressions are complex enough, to do so. Alternatively, if an automatic parallelization of the program is possible based on techniques capable of predicting the granularity of the subexpressions, the proposed language can serve as the intermediate language to which these tools translate.[1]

First, we will extend the syntax of expressions with what will be termed a letpar-expression:

$$\text{e} ::= \text{letpar } X_1 := e_1 \ \& \ \ldots \& \ X_n := e_n \text{ in } e$$

where e_i has to be an application of a defined function to terms ($1 \leq i \leq n$) and e may contain the new auxiliary variables $X_1, \ldots, X_n$. To simplify the implementation, we allow at most one letpar-expression in each rule. This is no real restriction, since it is always possible to decompose an expression by introducing new functions.

The meaning of a letpar-expression is that $e_1,\ldots,e_n$, and e (!) will be narrowed in parallel. e will be evaluated using a leftmost innermost strategy. If e needs the outcome of some e_i, it has to wait (suspension) until the corresponding process is ready (synchronization), at which point it can continue (reactivation). The outcome of e_i is *needed* if an X_i or a variable Y, occurring in e and e_i, is accessed during the narrowing of e. In order to facilitate backtracking (to be discussed in Section 4), we will require that the outcomes of $e_1, \ldots, e_n$ are always needed in the same order. If some outcomes are needed in a runtime dependent order, we say that all of them are needed, when the first of them is needed. For example in

$$\text{letpar } X_1 := e_1 \ \& \ X_2 := e_2 \text{ in h } (b \rightarrow X_1 \ \Box \ X_2) \ X_1 \ X_2$$

the outcomes of e_1 and e_2 are both needed after the narrowing of b.

In order to make the parallelism in the *towers* example explicit, we can change the function *towers* as follows:

fun *towers*: pin → pin → pin → nat → (list (list pin)) → (list (list pin)) → bool.
    towers Source Help Dest 1 L R :=   R = [[Source, Dest] | L].
    towers Source Help Dest (suc M) L R :=
        letpar L_1 := *towers* Help Source Dest M L R_1 &
            L_2 := *towers* Source Dest Help M X R
        in [[Source,Dest] | R_1] = X
        → true.

Note that the type of *towers* had to be changed in order to enable the decomposition of the nested application of *towers* using the free variable X. Remember that free variables are only allowed in the guard. The goal has to be changed correspondingly:

solve *towers* a b c N [ ] R.

R will be bound to the old result.

2.2.2 Independent And-Parallelism.

Even if two expressions are of adequate granularity there is still another issue, the existence of dependencies among the expressions to be executed in parallel, which may make it not always advisable to do so. In general, if two expressions which are narrowed in parallel have an unbound variable in common, this may lead to problems. Consider the following simple rules:

$$f \ a \ := g \ a \qquad f \ b \ := \ 1 \qquad g \ b \ := \ 1$$

If the goal is $f \ X \ = \ g \ X$, the two expressions $e_1 := f \ X$ and $e_2 := g \ X$ can be narrowed in parallel. Suppose that the narrowing of e_1 binds X to a and reduces $f \ a$ to $g \ a$. Now, the narrowing for e_2 may want to try the rule for g. This fails because X is already bound to a and it cannot be bound to b. After this, the narrowing of e_1 fails because there is no rule for $g \ a$. Hence, X is unbound and later bound to b, in order to try the second rule for f. But this is already too late for the narrowing of e_2, since it has already discarded the rule for g.

[1] This is the approach taken in the &-Prolog system [9], which has allowed a concurrent but rather independent development of the parallel language implementation and the automatic parallelization tools.

This example shows only one possible problem which may arise when dependent expressions are narrowed in parallel. Other more complicated situations can occur. A general solution ensuring that the correct result is always computed would arguably require a very complicated backtracking mechanism which would be a source of large overheads. Furthermore, there is an additional complication related to efficiency in ensuring that the search space to be explored by the program is not enlarged [10]. Due to these problems we take the approach, inspired by the theoretical results obtained for independent and-parallelism in Prolog [10] (ensuring correctness and "no-slowdown" properties for parallel execution), that subexpressions will only be narrowed in parallel if they are *independent*, i.e. they do not share unbound variables. As in [10], we offer two built-in (pseudo-)functions

$$ground\colon \alpha \to \text{bool}. \qquad\qquad independent\colon \alpha \to \beta \to \text{bool}.$$

ground e delivers the result *true*, if e is a ground term (i.e. a term without unbound variables) and *false* otherwise. *independent* e e' delivers the result *true*, if e and e' do not contain a common unbound variable and *false* otherwise. As in [3,8] the implementation of *ground* and *independent* may decide to deliver the value *false*, if it is too expensive to compute the exact value, e.g. if *ground* is applied to a very long list. These pseudo-functions may be used to guard a letpar-expression:

$$\text{e} ::= \text{letpar } grd \Rightarrow X_1 := e_1 \;\&\; \ldots \& \; X_n := e_n \text{ in } e$$

where *grd* is a conjunction of pseudo-function applications. The meaning of such a letpar-expression is that e_1, …, e_n, and e will be narrowed in parallel, if *grd* evaluates to *true*. Otherwise, they will be evaluated sequentially.

Accordingly, the second rule of the *towers* example might be modified to:

towers Source Help Dest (suc M) L R :=
    letpar (*ground* [Source, Dest, Help, M]), (*independent* L R) ⇒
        L_1 := *towers* Help Source Dest M L R_1,
        L_2 := *towers* Source Dest Help M X R
    in [[Source,Dest] | R_1] = X → true.

For the goal "*towers* a b c N [ ] R", this results in a sequential evaluation, since N is not ground. But e.g. "*towers* a b c (suc 1) [ ] R" would be narrowed in parallel.

In many cases, global program analysis [17] can detect that a variable is always bound to a ground term or that two variables are always independent. In those cases, the corresponding checks can be omitted at runtime. For the second goal, for instance, and for a given query pattern, it is possible to infer that Source, Dest, Help, and M are always bound to ground terms and that L and R will be independent, and all checks can be removed. Note that this type of program analysis is easier in Babel than in Prolog, due to the presence of types and nested function applications in Babel. In Prolog, nested function applications need to be flattened by introducing new variables, and it may take some effort to rediscover the structure which the programmer originally had in mind.

2.3 Deterministic Computations

In order to apply a rule, the arguments of a function are unified with the corresponding patterns on the left hand side and the guard is evaluated. Often, no variables of the rule are bound during the unification and the evaluation of the guard. We call this a *deterministic computation* [16,12], since the syntactic restrictions on Babel rules (Section 2) imply that no other rule for the same function needs to be tried. Hence, no choice point is needed in this case. Note that this optimization is more difficult in Prolog, since it uses relations rather than functions.

3 The And-Parallel Babel Machine

The PBAM is a parallel abstract graph narrowing machine that has been designed for the implementation of Babel on a shared memory multiprocessor. It has a decentralized organization of the runtime structures in a graph instead of a stack.

The graph component of the PBAM contains, among others, so-called task nodes which correspond to ordinary activation records of function calls, but which for decentralization purposes contain all the information needed to execute a function call, e.g. a stack for data manipulations and a local program counter. Some information for parallel execution is also needed.

The store of the PBAM consists of these components:

- Task Node:

TASK	code address	argument list	local variables
program counter		local stack	father pointer
backtracking information		parallelism information	

backtracking information:

local trail	determ. flag	backtracking pointer	last descendant pointer
backtracking address		program counter and local stack of the father	

parallelism information:

parallelism flag	T_1	state T_1	...	T_n	state T_n

- Constructor Node:

CONSTR	constructor name	list of components

- Variable Nodes: unbound: UBV or bound: VAR | Graph-address

Figure 1: Structure of graph nodes.

- the *program store* containing the translation of the Babel rules into PBAM code,
- the *graph*, which may contain task-, variable-, and constructor nodes,
- for each processor, one *active task pointer*, which points to the task node corresponding to the currently executed function call,
- a task queue, containing pointers to task nodes, ready to be executed.

3.1 The Graph

The graph consists of several nodes which are accessed via their addresses. Figure 1 shows the structure of such nodes. The computation is controlled by the *task nodes*, which represent function applications. Each task node contains:

- the *code address* of the code for the corresponding function symbol,
- pointers to the graph representations of the *arguments*,
- a list of pointers to initially unbound variable nodes representing *local variables*,
- the *program counter*,
- the *local stack*, needed for accessing and building structures in the graph,
- the *father pointer*, pointing to the node by which the current node has been created (this pointer is used when a task finishes successfully and control has to be returned to the father task), and
- the *return flag* indicating whether the task is looking for its first solution.

The task nodes contain also certain *backtracking* information. This consists of

- a *local trail* which keeps track of variable bindings to be removed in case of backtracking,
- a *determinism flag* used to discover whether the current function call is a deterministic computation,
- a *backtracking pointer* indicating the task that has to be forced to backtrack in case of a failure,
- a *last descendant pointer* indicating the last successful descendant of the current task (used to initialize the backtracking pointer of newly generated subtasks),
- a *backtracking address*, which is the program address of the next alternative rule of the current function call, and, finally,
- safe copies of the *program counter* and *the local stack of the father task* which have to be restored upon successful termination after backtracking has occurred.

All this will be explained in more detail in the following section, where the organization of backtracking is shown.

In order to control parallel execution, task nodes contain also some *parallelism* information. This consists of

- a *parallelism tag* indicating whether subtasks shall be evaluated in parallel,
- pointers to the tasks involved in the parallel execution,
- a *status tag* for each of these tasks, which should contain the values READY, DETERMINISTIC, EXECUTING, or DORMANT.

In addition to the task nodes, the graph contains *constructor* and *variable nodes*. *Constructor nodes* represent structured data. They contain the constructor name and a list of pointers to the graphs representing the components of the structure.

Variable nodes are needed for the organization of unification. We distinguish nodes for unbound and bound variables. Unbound variable nodes only consist of the tag (UBV). When a variable is bound, the corresponding node is changed into a bound variable node pointing to the node representing its binding.

3.2 Translation

Next, we show how a Babel program is translated into PBAM-code and sketch the behaviour of the machine instructions. Due to lack of space, we will only treat the translation of a letpar expression in detail. The general translation scheme will be

sketched using an example. An expression:

$$\text{letpar } grd \Rightarrow X_1 := f_1\ t_{1,1} \ldots t_{1,m_1}\ \& \ldots \&\ X_n := f_n\ t_{n,1} \ldots t_{n,m_n} \text{ in } e$$

produces the following code

code for grd
code for $\mathrm{t}_{1,1} \ldots \mathrm{t}_{1,m_1}$
PCALL (ca(f_1), m_1, k_1, 1)
$\vdots$
code for $\mathrm{t}_{n,1} \ldots \mathrm{t}_{n,m_n}$
PCALL (ca(f_n), m_n, k_n, n)
code for e

where k_i is the number of local variables of f_i $(1 \leq i \leq n)$.

grd is some test for independence and is translated by using the instructions GROUND and INDEPDNT. The code for evaluating the expression e has the following properties:

- Before the first appearance of each X_i or of a variable Y occurring in the terms $t_{i,1}, \ldots, t_{i,m_i}$, where Y is not guaranteed to be bound to a term, the instruction WAIT i is inserted.
- Each consultation of X_i is done by the instruction LOADR i.

In Figure 2, the translation of the *towers* example and of the goal (*towers* a b c (suc 1) [ ] R) is given. The first CALL instruction generates a task node for the goal and starts its evaluation. After a successful evaluation, the programmer is asked by the instruction MORE, whether more solutions are wanted. If this is the case, the FORCE instruction is executed and the task for the goal is forced to backtrack. Otherwise, control jumps to the STOP instruction at the end of the code. The translation of the functions follows after this preliminary code.

The *towers* example contains only one function with two rules. Since the first rule does not contain any parallelism, we omit its code due to the lack of space. The code for the second rule starts with an UNDO instruction that deletes the bindings that should have been produced by the first rule. Then, the backtracking address *fail* is stored in the actual task node (TRY_ME_ELSE *fail*). This label will be used if the rule fails. After this, the non-variable arguments of the actual task are unified with the corresponding terms on the left hand side of the rule. The 4th argument must consist of an application of the unary constructor *suc*. If this is the case, the UNIFYCSTR command leaves a pointer to the substructure (argument) of this argument on the stack. It is unified with the corresponding part of the term on the left hand side of the rule (UNIFYVAR). If the 4th argument is an unbound variable, this variable is bound to the appropriate term (by the three

```
         CALL (goal, 0, 1)
         MORE
         JPF stop
force:   FORCE
towers:  TRY_ME_ELSE rule2
         ... code for the first rule
         RET
rule2 :  UNDO
         TRY_ME_ELSE fail
         LOAD 4
         UNIFYCSTR (suc,1,bind)
         UNIFYVAR 1
         JMP rhs_2
bind:    LOADX 1
         CNODE (suc,1)
         BIND
rhs_2:   CUT fail
         GROUND 1
         GROUND 3
         GROUND 2
         GROUNDX 1
         INDEPDNT (5,6)
         LOAD 2
         LOAD 1
         LOAD 3
         LOADX 1
         LOAD 5
         LOADX 2
         PCALL (towers,6,3,1)
         LOAD 1
         LOAD 3
         LOAD 2
         LOADX 1
         LOADX 3
         LOAD 6
         PCALL (towers,6,3,2)
         LOAD 1
         LOAD 3
         CNODE (nil,0)
         CNODE (cons,2)
         CNODE (cons,2)
         WAIT 1
         LOADX 2
         CNODE (cons,2)
         WAIT 2
         LOADX 3
         CHECKEQ
         RET
fail:    UNDO
         FAILRET
goal:    TRY_ME_ELSE end
         CNODE (a,0)
         CNODE (b,0)
         CNODE (c,0)
         CNODE (1,0)
         CNODE (suc,1)
         CNODE (nil,0)
         LOADX 1
         CALL (towers,6,3)
         RETURNRESULT
end:     PRINTFAILURE
stop:    STOP
```

Figure 2: PBAM-code for the towers example.

commands following *bind*). These possible actions correspond to the read and write mode of the WAM [24], respectively. If the 4th argument starts with another constructor than *suc* the rule fails.

The translation of the body of the rule starts at label *rhs_2*. The CUT instruction at the beginning of the code for the body is used to detect deterministic computations, as mentioned in Subsection 2.3. If the current computation is deterministic this instruction sets the backtracking address to the fail label of the considered function. The technical details of the CUT instruction can be found in [12,16].

The top expression on the right hand side is a letpar construction. First, code is generated to check the groundness of the arguments Source, Dest, and Help (using GROUND instructions) and of the local variable M (using GROUNDX 1). Then, code to check the independence of L and R (instruction INDEPDNT (5,6)) is appended. GROUND, GROUNDX, and INDEPDNT set the parallelism flag (initially *true*) to *false*, if the corresponding check fails. If the parallelism flag is *false*, subsequent GROUND, INDEPDNT, and WAIT instructions will be ignored and PCALLs are handled like CALLs.

Using LOAD (to load an argument) and LOADX instructions (to load a local variable) the arguments for the call to towers are pushed on the stack. A task for the first call to *towers* is produced with a PCALL instruction, moving the arguments to the new task node. The second call to *towers* is handled analogously. The code for the main expression in the letpar, an equality, follows next. First the arguments of the equality are constructed on the stack using LOAD, LOADX, and CNODE instructions. CNODE (c,k) takes k components from the stack and inserts them into a new constructor node for constructor c.

However, before the use of R_1 (LOADX 2), the first parallel task needs to be finished. The instruction WAIT 1 is used to wait for this event. Similarly, WAIT 2 is placed before the use of X (LOADX 3).

Finally, control is given back to the calling task by RET. If the second and last rule also fails a jump to the command referenced by the current backtrack address (*fail*) is performed. All bindings produced by the rule are deleted (by UNDO) and backtracking is initiated (by FAILRET).

The translation of the goal appears after the code for the *towers* function. This translation is done analogously to the translation of the right hand side of a rule. A RETURNRESULT is included at the end in order to print the result after a successful termination. This instruction terminates the evaluation of the goal and gives control back to the top level task which continues with the MORE instruction. If the evaluation of the goal ultimately fails this is reported by the PRINTFAILURE command.

4 Backtracking

This section describes the management of backtracking both in the sequential and in the parallel case. Although the components of the backtracking information have already been sketched previously, we now explain their behaviour in detail. The local *trail* is a list of graph addresses indicating bound variable nodes, which have to be replaced by unbound variable nodes when an UNDO instruction is executed. The

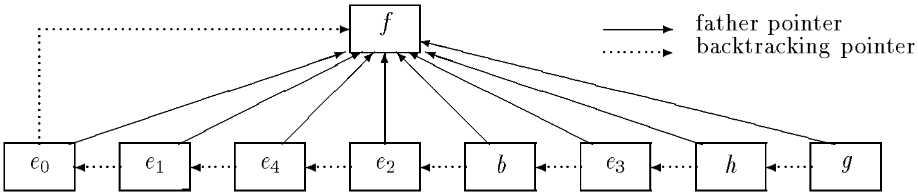

Figure 3: Structure of the graph.

management of backtracking is mainly controlled by two pointers: a so-called *backtracking pointer* to the *predecessor task* and a so-called *last descendant pointer* to the most recently generated task, i.e. the last successor task generated by the task or one of its descendants.

The notion of a *predecessor task* needs a precise definition. If a task is the first child of its father the predecessor task is the father. Otherwise, in the sequential case, it is the last task that terminated successfully before the currently executed task was generated.

In the parallel case, the predecessor task is initially the task finished before the code for the letpar was started. When WAIT i is executed, the backtracking pointer of the i-th parallel task is set to (the last descendant of) the previously finished task (if it is a sequential one), or to the $(i-1)$-th parallel task, if there is no call to a task between the use of task $i-1$ and i, i.e. between WAIT $i-1$ and WAIT i. Furthermore, the last descendant pointer of the father is updated with the last descendant pointer of the parallel task. By using this mechanism, the last descendant pointer of a task contains the most recently used task and it can help to set the backtracking pointer of the next task.

The backtracking pointers determine an implicit stack of nodes that reflects the order in which the nodes have been activated. While the father pointers are used to control the forward computation, the backtracking pointers determine the order of backward computation:

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

Here, it is important to point out that a deterministic task without children cannot give any new result, and it is not included in the backtracking chain. Any parallel task of this kind has the status DETERMINISTIC.

Some additional information for handling backtracking is needed in the task nodes. The *backtracking address* is the program address where the task must continue in case of backtracking (i.e. the address of the next rule). The backtracking information contains also a safe copy of the state of the father: a copy of the program counter and the stack. It is used to restore the state of the father in case of termination of a subtask after some backtracking has occurred. In this case the father must redo all evaluation that was done after the previous successful termination of the subtask. The copy is made during the execution of the CALL or PCALL command and the restoration is made after the successful reevaluation of a subtask, by the RET instruction in the sequential case and by the WAIT instruction in the parallel case. This situation is detected by checking the *return flag*.

Now, we can explain an instance of backward execution using an example. We want to evaluate the body of the rule:

$$f\ X\ := \ g\ e_0\ (\text{letpar}\ X_1 := e_1\ \&\ X_2 := e_2\ \&\ X_3 := e_3\ \ \text{in}\ (b\ X_1\ e_4\ X_2\ \rightarrow\ h\ X_3))$$

The structure of the graph after the evaluation of this expression is shown in Fig. 3.[2]

There are three situations when a task fails:

Sequential backtracking to the predecessor task appears, if the failing task was not spawned by a letpar (e.g. when backtracking from g to h, h to e_3, b to e_2 (e_3 may still be running), e_4 to e_1, and e_0 to f).

Parallel inside backtracking occurs when a task fails during its execution in parallel with other tasks. In this case, *intelligent backtracking* can be performed. Since we use independent and-parallelism, the failure is certainly not caused by a bad binding of a parallel sibling. Hence, the parallel sibling tasks and the intermediate sequential tasks (like e_4 and b) are removed and their bindings are undone (by the FAILRET instruction of the failing task). In our example, this kind of backtracking occurs if e_1, e_2 or e_3 fail, looking for their first solution. They will backtrack to e_0.

[2]Note that the order in which the expressions are linked in the backtracking chain is the order in which they are used, rather than that in which they are generated or specified.

Sequential inside backtracking occurs when the reevaluation of an originally parallel task T_i fails. If it is not the first one, it backtracks to the predecessor task like in sequential backtracking (e.g. from e_3 to b, e_2 to e_4), but it is converted into a dormant task (which can be reexecuted). If it is the first one, the complete letpar expression fails and all the parallel tasks are disposed. If T_i returns with success, the rest of the dormant tasks are evaluated from scratch (in parallel).

Returning to the example: if the reevaluation of e_1 fails, the whole letpar expression fails and all the parallel tasks e_1, e_2, and e_3 are deallocated. However, when a new solution is generated, e_2 and e_3 are started again, and e_4 is executed. After this, the corresponding WAIT command before the use of X_2 ensures the synchronization with e_2 and the execution of b is started. The synchronization with e_3 can wait until b finishes.

Note that e_2 and e_3 are only restarted when e_4 has produced a new result. The policy is to restart the execution of parallel tasks only in the case of a forward execution and not in a backward one.[3]

Recall that deterministic tasks are not included in the backtracking chain. Therefore, they are not affected by the backtracking mechanism and, hence, they are not reevaluated. However, the result is kept for later use. Suppose that, in our example, e_2 is a deterministic task. A failure in b yields backtracking to e_4, but the result of e_2 can be accessed after a successful reevaluation of e_4.

5 Some Related Work

Due to length limitations we will only review some directly related work. As mentioned in the introduction, the approach taken herein contrasts with that of other languages, normally termed *Logic functional languages*, which are based on adding support for functional characteristics to a logic language both at the language and implementation levels. A notable example of such languages is K-LEAF [1]. A parallel execution model for this system has also been proposed and implemented. However, it is based on the exploitation of or-parallelism and its abstract machine is an extension of a logic abstract machine, rather than a functional one. Thus, the solutions presented herein are different and complementary to those presented for K-LEAF. Since there is also backtracking in Babel, or-parallelism can also be exploited by extending the PBAM with either the environment sharing or the environment copying techniques used in the combination of and- and or-parallelism in Prolog, or, perhaps, with the techniques used in K-LEAF.

Some authors have proposed or implemented functional logic systems which support dependent And-parallelism (e.g. [13]). In [13] every variable belongs to a process, and only the owner may bind it. Usually, the extended synchronization in such systems imposes more overhead. It is not clear at this point whether this overhead would be worthwhile. An interesting logic programming system which does support dependent and-parallelism is the Andorra-I system [4], which allows (in addition to or-parallelism) dependent and-parallelism but only for goals which can be found to be deterministic. A similar approach could be taken in the PBAM, with the advantage of being able to more easily detect determinism. This deterministic dependent and-parallelism could be combined with the non-deterministic independent and-parallelism presented in the spirit of the combination of the Andorra-I and &-Prolog systems presented in the IDIOM model [22].

The approach presented in this paper borrows similarities from the independent and-parallelism found in Prolog [3,6,9,10] and exploited in &-Prolog. The main difference is that in that scheme the processes for some expressions e_1 & … & e_n are not linked by an expression e in a letpar expression. Rather, the evaluation of e would have been placed after that of $e_1, \ldots, e_n$. This leads to a simpler backtracking mechanism, but it requires the complete reevaluation of e, if some e_i has to produce a new solution. Many of the compile-time techniques developed in the context of this work could be extended to be useful for the model presented in this paper.

The idea of running $e_1, \ldots, e_n$, and e in parallel, and to synchronize e and e_i, when e needs the result of e_i, was used for the implementation of a purely functional language in [15]. In their framework, the order of the processes is not important, since in a functional language there is no backtracking.

The idea of allowing parallel expressions to proceed until their results are required, although developed independently, is reminiscent of Halstead's multilisp "future" construct [5]. However, the reasons for which synchronization is imposed are quite different in Babel, since they are related to nondeterminism and the partly logical nature of the language.

[3] This corresponds to "point backtracking" as opposed to "streak backtracking" [6].

6 Conclusions and Future Work

We have presented some techniques for the parallel implementation of narrowing, by integrating mechanisms used in functional and logic programming implementations. We propose a new synchronization model, where it is only necessary to wait for the value of a parallel subexpression, if the result is needed. The approach increases the parallelism and decreases the reevaluation effort, especially in the case of deterministic computations. We have also presented an abstract machine, the PBAM, capable of implementing the model of parallelism, sketched the translation process from Babel to PBAM instructions, and discussed in detail the more involved issues, such as backtracking.

Currently we are developing a concrete Babel implementation on a Sequent shared-memory multiprocessor, based on the approach presented. We hope to have a complete running prototype and some performance measurements in the near future. As future work, we plan to investigate the efficient management of the stopping of processes in order to implement the intelligent backtracking, and efficient groundness and independence tests. We also plan to develop a parallel machine for lazy narrowing, extending the design presented in [18]. Moreover, we wish to integrate the machine in a distributed environment. The graph structure will support this.

6.0.1 Acknowledgements.

We thank all the members of the Babel teams in Aachen and Madrid as well as the anonymous referees for helpful comments. Manuel Hermenegildo is supported in part by ESPRIT project “PEPMA” and CICYT project TIC90-1105-CE.

References

[1] P. Bosco, C. Cecchi, C. Moiso, M. Porta, G. Sofi: Logic and Functional Programming on Distributed Memory Architectures, Proc. 7th ICLP, 325-339,(1990).

[2] S. Debray, N.-W. Lin, M. Hermenegildo: Task Granularity Analysis in Logic Programs, Proc. ACM Conf. on Programming Language Design and Implementation, 1990.

[3] D. DeGroot: Restricted And-parallelism, Conf. on 5th Generation Comp. Syst., 1984.

[4] G. Gupta, V. Santos Costa, R. Yang, M.V. Hermenegildo: IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism, Proc. of the 1991 Int'l. Logic Programming Symposium, MIT press.

[5] R. Halstead: Multilisp: A Language for Concurrent Symbolic Computation, ACM Trans. on Prog. Languages and Systems 7:4, October 1985, pp. 501-538

[6] M.V. Hermenegildo: An Abstract Machine for Restricted And Parallel Execution of Logic Programs, 3rd Int. Conf. on Logic Programming, LNCS 225, 25-39 (1986).

[7] M.V. Hermenegildo, R.I. Nasr: Efficient Management of Backtracking in And-parallelism, 3rd Int. Conf. on Logic Programming, LNCS 225, 40-50 (1986).

[8] M.V. Hermenegildo, M. Carro: Experimenting with Independent And-Parallel Prolog using Standard Prolog, Proc. PRODE'91, and Tech. Report, UP Madrid.

[9] M.V. Hermenegildo, K.J. Green: &-Prolog and its performance: Exploiting Independent And-Parallelism, Proc. 7th ICLP, 253-268, (1990).

[10] M. Hermenegildo, F. Rossi: Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions, Journal of Logic Programming, to appear (1992).

[11] H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodríguez Artalejo: Graph-Based Implementation of a Functional Logic Language, ESOP, LNCS 432:271–290 (1990).

[12] H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodríguez Artalejo: Graph-Narrowing to Implement a Functional Logic Language, Technical Report, UP Madrid (1992).

[13] H. Kuchen, W. Hans: An And-Parallel Implementation of the Functional Logic Language Babel, Aachener Informatik-Bericht 91/12:119–139, RWTH Aachen (1991).

[14] A. King and P. Soper, Granularity Analysis of Concurrent Logic Programs, 5th Int. Symp. on Computer and Information Sciences, Nevsehir, Turkey, 1990.

[15] R. Loogen, H. Kuchen, K. Indermark, W. Damm: Distributed Implementation of Programmed Graph Reduction, PARLE, LNCS 365:136–157(1989).

[16] R. Loogen, S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages, 3rd PLILP, LNCS 528:335–346 (1991).

[17] K. Muthukumar, M.V. Hermenegildo: The CDG, UDG and MEL methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism, Proc. ICLP, (1989).

[18] J.J. Moreno Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, 2nd ALP, LNCS 456:298–317 (1990).

[19] J.J. Moreno Navarro, M. Rodríguez-Artalejo: Babel: A Functional and Logic Programming Language Based on Constructor Discipline and Narrowing, Int. Conf. on Algebraic and Logic Programming (ALP), LNCS 343:223–232 (1989).

[20] J.J. Moreno Navarro, M. Rodríguez Artalejo: Logic Programming with Functions and Predicates: The Language Babel, J. of Logic Programming:12: 191–223 (1992).

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

[22] V. Santos Costa, D.H.D. Warren, R. Yang: Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism, Proc. Principles and Practices of Parallel Programming, to appear.

[23] D.A. Turner: Miranda: A Non-Strict Functional Language with Polymorphic Types, ACM Conf. on Functional Languages and Computer Arch., LNCS 201:1–16 (1985).

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