Result Directed Computing in a Functional Logic Language

Herbert Kuchen
RWTH Aachen*

Francisco Javier López-Fraguas
Universidad Complutense de Madrid†

Abstract

In a functional logic language, especially in a constraint functional logic language, often the topmost constructor is known which an expression should be evaluated to. E.g. in a guarded expression $b \rightarrow e$, b should only be evaluated to *true*. Evaluating b to *false* would only waste resources and risk to run into an infinite computation. Hence, some kind of result directed computation can be used to reduce the search space by cutting uninteresting computations. One possibility to achieve this result direction is to integrate it into the abstract machine which is the basis for the runtime system. Here, we show how to achieve result directed computation by means of a program transformation. A runtime system without special facilities for result directed computing can be used to run the transformed programs. Thus, runtime overhead for result directed computing is avoided.

1 Introduction

During the last years, many proposals for combining the functional and logic programming paradigms have been made [BL86,DL86]. In particular, so called *functional logic languages* [Re85] retain functional syntax but use *narrowing* as operational semantics. Narrowing is a unification based parameter passing mechanism which subsumes rewriting and SLD resolution.

There are also several works aiming at the efficient implementation of logic + functional languages by means of abstract machines supporting combinations of functional and logic programming capabilities [BBCMMS89,BCGMP89,Ha90,KLMR90,MKLR90,Mu90,Lo91].

In the following, we will concentrate on the functional logic language Babel [MR88] [MR92]. One feature of Babel which is of special relevance to this paper is that boolean functions may, besides the value *true*, also return the value *false*. In particular, the rules for the predefined functions $\wedge,\vee,\neg,=$ consider the two truth-values, both as arguments and as results. This provides more flexibility, allowing the user to write clearer and more concise programs in many cases. This increase in the expressivity of boolean functions may have a significant computational cost if only one value contributes to a successful computation, but we do not avoid computations, which produce the other value.

In [KLMR92b], a new operational semantics for Babel and an implementation for it were proposed. The main idea is to allow disequalities $X \neq t$ as answers. In general, such disequations cannot be replaced by any equivalent, finite set of equations. The operational semantics combines lazy narrowing with equality and disequality constraint solving. While solving these constraints, only one result, either *true* or *false*, is interesting. Thus, especially for the efficient implementation of the new operational semantics, it is crucial to find mechanisms which avoid useless results.

Of course, result directed computing can be straightforwardly generalized to arbitrary (constructed) data structures. Now, only computations are interesting, where the result starts with some desired topmost constructor.

One possibility to achieve result directed computing is to integrate it into the abstract machine which is the basis for the runtime system. One step in this direction is the tag mechanism included in the abstract operational semantics, presented in [KLMR92b].

In the present paper, we show two program transformations which ensure that the transformed program runs result directed on the existing narrowing machines. This approach avoids runtime overhead for result directed computing. The idea is to replace a rule by a set of specialized versions of it. These specialized versions are generated by using the information about the desired result in order to apply partial evaluation techniques [BEJ88] to the right hand side of the rule.

*Lehrstuhl f. Informatik II, Ahornstr. 55, D-5100 Aachen, Germany, herbert@zeus.informatik.rwth-aachen.de
†Departamento de Informática y Automática, Av. Complutense s/n, 28040 Madrid, Spain, email: fraguas@emducm11.bitnet

The rest of the paper is organized as follows. In Section 2, we describe the syntax and semantics of Babel. In Section 3, we present the mentioned program transformations. Section 4 contains some experimental results. A conclusion is found in Section 5.

2 Babel

The language Babel integrates functional and logic programming in a flexible and mathematically well-founded way. It is based on a constructor discipline and uses narrowing as evaluation mechanism [Re85]. Babel has a polymorphic type system, similar to that of [Mi78]. First, we will only use the first order fragment of Babel. Higher order functions will be discussed later. Also, rather than giving a precise description of the syntax and semantics of Babel (which can be found in [MR88,MR92,KLMR90,KLMR92]) we will first show a small example program and then present some basic concepts:

datatype name := *mary* | *paul* | *susan* | *maud* | *bill* | *jack* | *jane* | *margret* | *henry* |
elizabeth | *sam*.
datatype bool := *true* | *false*.
datatype list α := *nil* | *cons* α (*list* α).

fun father: *name* $\rightarrow$ *name*.
father mary := *bill*.
father paul := *bill*.
father maud := *henry*.
father bill := *sam*.
father jack := *henry*.
father jane := *sam*.
father susan := *jack*.

fun mother: *name* $\rightarrow$ *name*.
mother mary := *maud*.
mother paul := *maud*.
mother maud := *margret*.
mother bill := *elizabeth*.
mother jack := *margret*.
mother jane := *elizabeth*.
mother susan := *jane*.

fun male: *name* $\rightarrow$ *bool*.
male mary := *false*.
male maud := *false*.
male bill := *true*.
male jack := *true*.
male jane := *false*.
male susan := *false*.
...

fun parents: *list name* $\rightarrow$ *list name*.
parents [ ] := [ ].
parents [X | L] := [*mother* X, *father* X | *parents* L].

fun disjoint: *list* α $\rightarrow$ *list* α $\rightarrow$ *bool*.
disjoint [ ] L := *true*.
disjoint [X | Xs] L := $\neg$(*member* X L) $\wedge$ *disjoint* Xs L.

fun member: α $\rightarrow$ *list* α $\rightarrow$ *bool*.
member X [ ] := *false*.
member X [Y | L] := $X = Y \rightarrow$ *true* $\Box$ *member* X L.

fun m_cousin: *name* $\rightarrow$ *name* $\rightarrow$ *bool*.
m_cousin X Y := *male* Y $\wedge$ *disjoint* (*parents* [X]) (*parents* [Y])
$\wedge$ $\neg$(*disjoint* (*parents* (*parents* [X])) (*parents* (*parents* [Y]))).

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 m_cousin susan X $\rightarrow$ *true*.

which would bind X to a male cousin of *susan* and return *true*. Remark that *member*, *disjoint* and *m_cousin* may deliver the results *true* or *false*, and that *m_cousin* makes use of both results for *disjoint*.

A *datatype* definition introduces a new ranked *type constructor* (e.g. *name*, *bool* (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 *function types* of the form $\tau \rightarrow \tau'$ and *type variables* $(\alpha, \beta, \ldots)$ representing any valid type) the resulting type expression is a valid type, e.g. "*list name*" and "*list* α". Such a type is called *constructed type*. 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 mary nil*) is of type "*list name*" (α is replaced by *name*).

A Prolog-like syntax for lists is allowed, i.e. $[e \mid e']$ 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

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

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:

$e ::= t$	% term
$\mid (c\ e_1 \ldots e_n)$	% application of a n-ary data constructor c ($n \geq 0$)
$\mid (f\ e_1 \ldots e_n)$	% application of a n-ary function f ($n \geq 0$)
$\mid (b \rightarrow e)$	% guarded expression, meaning "*if b then e else undefined*"
$\mid (b \rightarrow e_1 \Box e_2)$	% conditional expression, meaning "*if b then* e_1 *else* e_2"

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. Parentheses can be omitted, if the structure of an expression is clear.

Babel functions are functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is only one result. This is ensured by syntactic restrictions for the rules:

- Only the guard (not the body) may contain free variables, i.e. variables not occurring on the left hand side.
- For two different rules for the same function, at least one of the following holds:
  - their left hand sides are not unifiable,
  - their bodies are syntactically identical (after applying the most general unifier σ of the left hand sides), or
  - their guards are incompatible.

Incompatibility [KLMR92] is a decidable property of two boolean expressions b and b', guaranteeing that $(b\ \sigma\ \wedge\ b'\ \sigma)$ cannot be reduced to *true*. 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. Moreover, several basic (infix and mixfix) operations like equality (=), conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), $(\_ \rightarrow \_)$ (guarded expression), $(\_ \rightarrow \_\Box\_)$ (conditional expression), and so on are predefined, e.g.

$(true \rightarrow X) := X.$ $\qquad$ $(true \rightarrow X \Box Y) := X.$ $\qquad$ $(false \rightarrow X \Box Y) := Y.$

2.1 Narrowing Semantics

Here, we will present an operational semantics of Babel based on *narrowing*. To use narrowing as the operational semantics of (syntactically) functional languages was first proposed by Reddy [Re85]. Another, more expressive, semantics combining narrowing and constraint solving can be found in [KLMR92b].

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

$$(f\ e_1 \ldots e_i \ldots e_n) \Rightarrow_\sigma (f\ e_1\sigma \ldots e'_i \ldots e_1\sigma)$$
$$(c\ e_1 \ldots e_i \ldots e_n) \Rightarrow_\sigma (c\ e_1\sigma \ldots e'_i \ldots e_1\sigma)$$

where f is an n-ary defined function and c is an n-ary constructor.

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

In the present paper, we consider a lazy narrowing strategy. For an easy definition of lazy narrowing, it is convenient to assume a special class of programs, so-called *uniform Babel programs* [MKLR90]. In such programs, all the left hand sides of the rules for a given function have the same simple shape. If $f\ t_1 \dots t_n := e$ and $f\ s_1 \dots s_n := e'$ are two rules for the function f, then for each argument position $i = 1, \dots, n$, t_i and s_i are either both variables or both applications of a constructor to (zero or more) variables (!). In the latter case, we say that f demands the i-th argument. Using the lazy narrowing strategy, the arguments of a function f may only be narrowed, if they are demanded by f. An automatic transformation of general Babel programs to uniform programs is described in [MKLR90].

Alternatively, an innermost (eager) narrowing strategy [KLMR90] can be considered, where all arguments of a function have to be narrowed before a rule for the function can be applied. Most of the results, presented in the sequel, also hold for innermost narrowing and many other narrowing strategies (e.g. parallel narrowing strategies).

The goal (*member X L*) in the cousin example can be (lazily) narrowed as follows:

1.solution:	$member\ X\ L$	$\Rightarrow_{\{L/[\,]\}}$	$false$
2.solution:	$member\ X\ L$	$\Rightarrow_{\{L/[Y\mid L']\}}$	
	$X = Y \rightarrow true \ \Box\ member\ X\ L'$	$\Rightarrow_{\{Y/X\}}$	
	$true \rightarrow true \ \Box\ member\ X\ L'$	$\Rightarrow_{\{\,\}}$	$true$
3.solution:	$\dots$		

The second solution consists of the result *true* and the answer $\{L/[X \mid L']\}$.

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

3 The Program Transformation

In order to reach a result oriented computation, we will use two (alternative) program transformations, T_1 and T_2. First, we will describe T_1.

All the rules of a function f with type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ are transformed using the following scheme:

$$ruletrans(f\ t_1 \dots t_n := e) := \begin{cases} \{f_{any}\ t_1 \dots t_n := \mathcal{E}(e, any)\} & \text{if } \tau \text{ is a type variable} \\ \{f_c\ t_1 \dots t_n := \mathcal{E}(e, c) \mid c \in P(e) \cup \{any\},\ P(e) \neq \emptyset\} & \text{otherwise} \end{cases}$$

The new functions f_c introduced by *ruletrans* deliver the same result as f (for the same arguments) as long as this result has c as the topmost constructor. Otherwise, f_c is undefined. *ruletrans* uses the scheme $\mathcal{E}$ for the translation of expressions. $\mathcal{E}(e, c)$ transforms e in such a way that it may only deliver a result with topmost constructor c or fail. $c = any$ means that arbitrary results are possible. $P(e)$ (defined below) approximates the set of possible topmost constructors of the result of e. Note that even for f_{any}, which may produce arbitrary results, we achieve result directed computing in the right hand side. If the result of a function has a type variable as type, we renounce result directed computing, since we would have to generate a new rule for each constructor in the program. Moreover, result direction usually does not gain much in this case.

In order to improve the readability, we here present a simplified version of the $\mathcal{E}$ scheme. A more sophisticated variant can be found in the appendix. It simplifies the program using the information that the result has to start with the topmost constructor c. In particular, it simplifies the produced expression, if it is known that a subexpression will always fail, e.g. $\mathcal{E}(b, any) \rightarrow \mathcal{E}(e_1, c) \Box \mathcal{E}(e_2, c)$ is simplified to $\mathcal{E}(b, true) \rightarrow \mathcal{E}(e_1, c)$, if it can be inferred that e_2 cannot produce a result starting with the desired topmost constructor c. This can be further simplified to $fail$ (a nullary function without rules), if it is sure that b cannot produce the result *true*.

$$\mathcal{E}(e,\ c) := fail \qquad \text{if } c \notin P(e) \cup \{any\}$$

If the above failure rule cannot be applied, one of the following definitions will be used.

$$\mathcal{E}(f\ e_1 \dots e_n,\ c) := f_c\ \mathcal{E}(e_1, c_1) \dots \mathcal{E}(e_n, c_n) \qquad n \geq 0$$

$$\text{where } c_i := \begin{cases} c'_i, & \text{if every rule } f\ t_1 \dots t_n := e \text{ with } c \in P(e) \cup \{any\} \\ & \text{has } c'_i \text{ as the topmost constructor of } t_i \\ any, & \text{otherwise} \end{cases} \qquad (1 \leq i \leq n)$$

$$\mathcal{E}(c'\ e_1 \dots e_n,\ c) := c'\ \mathcal{E}(e_1, any) \dots \mathcal{E}(e_n, any), \quad \text{if } c = c' \text{ or } c = any,\ n \geq 0$$

$$\mathcal{E}(b \rightarrow e_1,\ c) := \ \mathcal{E}(b, true) \rightarrow \mathcal{E}(e_1, c)$$

$$\mathcal{E}(b \rightarrow e_1 \ \Box\ e_2,\ c) := \ \mathcal{E}(b, any) \rightarrow \mathcal{E}(e_1, c) \ \Box\ \mathcal{E}(e_2, c)$$

$$\mathcal{E}(e_1 = e_2,\ c) := \begin{cases} \mathcal{E}(e_1, c') = \mathcal{E}(e_2, c'), & \text{if } P(e_1) \cap P(e_2) = \{c'\},\ c = \mathit{true} \\ \mathit{fail}, & \text{if } P(e_1) \cap P(e_2) = \emptyset,\ c = \mathit{true} \\ \mathcal{E}(e_1, \mathit{any}) = \mathcal{E}(e_2, \mathit{any}), & \text{otherwise} \end{cases}$$

$$\mathcal{E}(b_1 \vee b_2,\ c) := \begin{cases} \mathcal{E}(b_1, \mathit{true}) \oplus \mathcal{E}(b_2, \mathit{true}), & \text{if } c = \mathit{true} \\ \mathcal{E}(b_1, c) \vee \mathcal{E}(b_2, c), & \text{otherwise} \end{cases}$$

$$\mathcal{E}(b_1 \wedge b_2,\ c) := \begin{cases} \mathcal{E}(b_1, \mathit{false}) \odot \mathcal{E}(b_2, \mathit{false}), & \text{if } c = \mathit{false} \\ \mathcal{E}(b_1, c) \wedge \mathcal{E}(b_2, c), & \text{otherwise} \end{cases}$$

$$\mathcal{E}(\neg b_1,\ c) := \neg \mathcal{E}(b_1, \bar{c}), \qquad \text{where } \bar{c} := \begin{cases} \mathit{true}, & \text{if } c = \mathit{false} \\ \mathit{false}, & \text{if } c = \mathit{true} \\ \mathit{any}, & \text{if } c = \mathit{any} \end{cases}$$

$$\mathcal{E}(X,\ c) :=\ X$$

Rules which are never used can be omitted. The auxiliary operations $\oplus$ and $\odot$ are predefined by the rules:

$$\begin{array}{lll} X \ \oplus\ Y \ :=\ X \to \mathit{true} & \qquad & X \ \odot\ Y \ :=\ \neg\, X \to \mathit{false} \\ X \ \oplus\ Y \ :=\ Y \to \mathit{true} & & X \ \odot\ Y \ :=\ \neg\, Y \to \mathit{false} \end{array}$$

Note that the transformation does not change variables. Hence, the result of a result directed computation is either a logical variable, or an expression starting with the desired constructor, or a failure. A more sophisticated handling of variables is described in Subsection 3.1.

$P(e)$ approximates (subsumes) the set of constructors, which can be the topmost constructors of the result after evaluating e. P is defined by recursion over the structure of the argument expression. Since a program typically contains recursive and mutually recursive definitions, a least fixed point has to be computed in order to define P. As usual, this least fixed point is computed iteratively. The termination of this iteration is ensured by the fact that there are only finitely many constructors in the program.

$P_0(e) := \emptyset$ for all expressions e

For $i \geq 0$, we define:

$$P_{i+1}(f\ e_1 \ldots e_n) := \bigcup_{e \in \mathit{prhs}(f)} P_i(e)$$

where $\mathit{prhs}(f)$ denotes the set of right hand sides of those rules $f\ t_1 \ldots t_n\ :=\ e$ for f with $P_{i+1}(t_j) \cap P_{i+1}(e_j) \neq \emptyset$ for all $j = 1, \ldots, n$.

$$P_{i+1}(c\ e_1 \ldots e_n) :=\ \{c\}$$

$$P_{i+1}(b \to e_1) := \begin{cases} P_{i+1}(e_1), & \text{if } \mathit{true} \in P_{i+1}(b) \\ \emptyset, & \text{otherwise} \end{cases}$$

$$P_{i+1}(b \to e_1 \Box e_2) := \begin{cases} P_{i+1}(e_1) \cup P_{i+1}(e_2) & \text{if } P_{i+1}(b) = \{\mathit{true}, \mathit{false}\} \\ P_{i+1}(e_1) & \text{if } P_{i+1}(b) = \{\mathit{true}\} \\ P_{i+1}(e_2) & \text{if } P_{i+1}(b) = \{\mathit{false}\} \\ \emptyset & \text{if } P_{i+1}(b) = \emptyset \end{cases}$$

$$P_{i+1}(X) := \{c \mid c \text{ constructs a term of the type of } X\}$$

If the type of X is a type variable, $P_{i+1}(X)$ will be the set of all constructors.

$$P_{i+1}(e_1 = e_2) := \begin{cases} \emptyset & \text{if } P_{i+1}(e_1) = \emptyset \text{ or } P_{i+1}(e_2) = \emptyset \\ \{\mathit{false}\} & \text{if } P_{i+1}(e_1) \cap P_{i+1}(e_2) = \emptyset, \\ & \quad P_{i+1}(e_1) \neq \emptyset,\ P_{i+1}(e_2) \neq \emptyset \\ \{\mathit{true},\ \mathit{false}\} & \text{otherwise} \end{cases}$$

$$P_{i+1}(b_1 \vee b_2) := ((P_{i+1}(b_1) \cup P_{i+1}(b_2)) \cap \{\mathit{true}\}) \cup (P_{i+1}(b_1) \cap P_{i+1}(b_2) \cap \{\mathit{false}\})$$

$$P_{i+1}(b_1 \wedge b_2) := ((P_{i+1}(b_1) \cup P_{i+1}(b_2)) \cap \{\mathit{false}\}) \cup (P_{i+1}(b_1) \cap P_{i+1}(b_2) \cap \{\mathit{true}\})$$

$$P_{i+1}(\neg b) := \{\bar{c} \mid c \in P_{i+1}(b)\}$$

$$P(e) := P_k(e),$$

where $k := min\{j \mid j > 0, (\forall e \in RHS)\ \ P_j(e) = P_{j-1}(e)\}$
and RHS is the set of right hand sides of all the rules in the considered program.

In order to use a transformed program, a goal e must be replaced by $\mathcal{E}(e,\ any)$.

3.1 Transforming Variables

An alternative treatment of variables could be:

$$\mathcal{E}(X,\ c) := \begin{cases} X = (c\ X_1 \ldots X_n) \rightarrow X, & \text{if } c \neq any, \text{ and } X_1, \ldots, X_n \text{ are "new" variables} \\ X, & \text{if } c = any \end{cases}$$

This would avoid logical variables as results of result directed computations, but in most cases, it would not gain anything, since the same binding would be introduced later while unifying the goal with the head of a rule. This expected unification is the reason that we are allowed to look for the considered topmost constructor.

Another possibility is to substitute X by $(c\ X_1 \ldots X_n)$ in the whole rule. Thus, we propagate the discovered restriction on the variable X to all its occurrences in the rule. In many cases this contributes to a further reduction of the number of computations. Unfortunately, this substitution cannot be safely applied in all cases. The following example reveals the problem. Consider the expression $\hat{e}$

$$\hat{e} \equiv p\ X \rightarrow\ f\ X \ \Box\ g\ (h_1\ X)\ (h_2\ X)$$

and suppose that f forces X to be of the form $(c\ Y)$. The problem is that $(p\ X)$ may evaluate to *false*. That means that, instead of $(f\ X)$, we have to evaluate $(g\ (h_1\ X)\ (h_2\ X))$, which may produce a different binding for X.

If, on the other hand, p forces X to be of the form $(c\ Y)$, we can safely substitute X by $(c\ Y)$ in the whole rule, since $(\_ \rightarrow \_\Box\_)$ demands its first argument and hence the evaluation of $(p\ X)$ cannot be avoided.

In order to make precise when the substitution can be applied, we need the notions of *demanded variables* of an expression and *demanded arguments* of a function. We borrow these notion from [HLW92], where demandedness information is used for deriving improved lazy narrowing strategies. Informally, a variable X is *demanded* by an expression e if $X \equiv e$, or if $e \equiv (f\ e_1 \ldots e_n)$ and X is demanded by some e_i $(1 \leq i \leq n)$ and f demands its i-th argument, i.e. all the rules for f demand the i-th argument. A rule $f\ t_1 \ldots t_n := e'$ *demands* the i-th argument, if t_i is not a variable or if t_i is a demanded variable of e'. A formal definition of these notions, given in terms of the least fixed point of a set of mutually recursive equations, can be found in [HLW92].

If we now encounter $\mathcal{E}(X, c)$ (where $c(\neq any)$ is an n-ary constructor, $n \geq 0$), while transforming a rule, and the variable X is demanded by the right hand side of this rule, then we substitute X by $(c\ X_1 \ldots X_n)$ in the (partially transformed) rule (where $X_1, \ldots, X_n$ are new variables).

If the variable X is not demanded by the right hand side *rhs*, one can decompose *rhs* by introducing a new auxiliary function h and replacing the maximal subexpression $\tilde{e}$ of *rhs* such that $\tilde{e}$ demands X by a call to h. h will have all variables occurring in $\tilde{e}$ as parameters. Suppose that, in the above example, h_1 requires X to have the form $(d\ X_1\ X_2)$ and that g demands its first argument. Then, $\hat{e}$ can be replaced by $(p\ X \rightarrow f\ X\ \Box\ h\ X)$ where h is defined by the rule:

$$h\ (d\ X_1\ X_2)\ \ :=\ \ g\ (h_1\ (d\ X_1\ X_2))\ (h_2\ (d\ X_1\ X_2))$$

3.2 An Example

If the example program of Section 2 is transformed using T_1, we get the program shown below. For simplicity, type declarations and some useless rules are omitted. Instead of indexing function symbols by *any*, we use the original symbols.

$father\ mary := bill.$	$mother\ mary := maud.$	$male_f\ mary := false.$	$male_t\ bill := true.$
$father\ paul := bill.$	$mother\ paul := maud.$	$male_f\ maud := false.$	$male_t\ jack := true.$
...	...	...	...

$$\begin{array}{ll} parents\ [\,] := & [\,]. \\ parents\ [X \mid L] := & [mother\ X,\ father\ X \mid parents\ L]. \end{array}$$

$$parents_{cons}\ [X \mid L] := [mother\ X,\ father\ X \mid parents\ L].$$

$$
\begin{array}{ll}
disjoint_t\ [\,]\ L := & true. \\
disjoint_t\ [X \mid Xs]\ \mathrm{L} := & \neg(member_f\ X\ L) \wedge disjoint_t\ Xs\ L. \\
 & \\
disjoint_f\ [X \mid Xs]\ L := & \neg(member_t\ X\ L) \odot disjoint_f\ Xs\ L. \\
 & \\
member_t\ X\ [Y \mid L] := & X = Y \rightarrow true \,\square\, member_t\ X\ L. \\
 & \\
member_f\ X\ [\,] := & false. \\
member_f\ X\ [Y \mid L] := & X = Y \rightarrow fail \,\square\, member_f\ X\ L. \\
 & \\
m\_cousin_t\ X\ Y := & male_t\ Y \wedge disjoint_t\ (parents\ [X])\ (parents\ [Y]) \\
 & \wedge \neg(disjoint_f\ (parents_{cons}\ (parents\ [X])) \\
 & \qquad (parents\ (parents\ [Y])). \\
 & \\
solve\ m\_cousin_t\ susan\ X \rightarrow true. &
\end{array}
$$

The second rule for $member_f$ can be improved using the full set of transformation rules shown in the appendix:

$$member_f\ X\ [Y \mid L] := \neg(X = Y) \rightarrow member_f\ X\ L.$$

3.3 Termination

The following example shows that the transformed program may terminate, while the original program runs into an infinite computation (type declarations are omitted):

original program:

$foo\ X := falsefct\ X \wedge \neg(X = 0) \rightarrow true.$
$foo\ X := X = 0 \rightarrow false.$
$falsefct\ 0 := false.$
$falsefct\ (suc\ X) := falsefct\ X.$
$solve\ foo\ X.$

transformed program:

$foo_{any}\ X := X = 0 \rightarrow false.$
$solve\ foo_{any}\ X.$

If we suppose that the rules are tried from top to bottom and a depth first evaluation strategy is used, then the original program will run into an infinite computation, since infinitely many narrowings of ($falsefct\ X$) will be tried. Actually, X will be bound to one natural number after the other, but all these bindings will lead to the undesired value *false*.

In the transformed program, the first rule for *foo* is deleted, since $P(falsefct\ X \wedge \neg(X = 0)) = \{false\}$ and hence $P(falsefct\ X \wedge \neg(X = 0) \rightarrow true) = \emptyset$. Thus, the second rule will be used, binding X to 0 and delivering the result *false*.

Thus, we have shown that the transformed program may terminate while the original one does not. On the other hand, it is clear from the construction that only useless computations are pruned. We only modify an expression in order to guarantee a specific topmost constructor of the result, if this was required by the context. Hence, it is clear that if the original program terminates, then the transformed program also terminates and delivers the same result.

3.4 Interaction with other Optimizations

Now, we want to examine, how our program transformation interacts with other optimizations which also aim at reducing the search space, in particular indexing and dynamic cut.

Indexing [Ai91,Wa83] is a well known technique whose purpose is to discard during runtime those rules for which it is known in advance that unification is going to fail. Indexing roughly works as follows. When a predicate/function is called, some preselected arguments are examined, and a jump to the first rule will be performed, where these arguments are unifiable with the corresponding arguments in the rule. Babel implementations also incorporate this feature.

The effect of indexing is not weakened by our program transformation. On the contrary, since a call to a function in the original program might have been replaced by a call to a more specialized version with less rules, there are more possibilities for useful indexing. In addition, if some variable in the left hand side of a rule is substituted by a term, the chances for indexing are furthermore increased.

Dynamic cut [KLMR92,LW91] is an important optimization incorporated in Babel implementations, making use of the fact that the rules for a function symbol really define a function, that is, different rules cannot yield different values for the same (ground) arguments. Consequently, the following mechanism (dynamic cut), which discards alternatives for a goal during runtime, is sound: if unification and evaluation of the guard of a given rule have been successful without binding any variable in the goal, then there is no need to try another rule for the considered function, if a later failure (or a request for another solution) forces backtracking.

Remark that this later failure may even occur when evaluating the body of the rule being tried. This may cause an unpleasant interaction of dynamic cut and our program transformation. Assume the following example program:

$$\begin{array}{llll} f\ X := & X = a\ \rightarrow\ \mathit{false}. & g\ X := & f\ X\ \rightarrow\ \mathit{true}. \\ f\ X := & \mathit{loop}\ X \wedge \neg(X = a) \rightarrow\ \mathit{true}. & g\ X := & X = a\ \rightarrow\ \mathit{true}. \end{array}$$

where *loop* is any function such that $(\mathit{loop}\ a)$ runs forever. Note that this is a non-ambiguous program.

For the goal $(g\ a)$, we obtain the result *true*, if a dynamic cut is used. Applying the first rule for f to $(f\ a)$ (which is called by the first rule for g) will give the value *false*. This forces backtracking because the guard (here $(f\ a)$) in a guarded expression needs to be evaluated to *true*). Since the second rule for f is discarded by a dynamic cut, the second rule for g is tried, returning the result *true*.

In this example, the program transformation detects that the call to $(f\ X)$ in the first rule for g should only use those rules for f which can yield the value *true*, i.e. only the second rule for f. But with this rule $(f\ a)$ runs forever.

Of course, it is easy to find an example where the program transformation behaves better than a dynamic cut. Unfortunately, combining both optimizations does not mean that all infinite computations are pruned which are eliminated by dynamic cuts alone. In general, a dynamic cut has two faces: detection of deterministic successful computations, and detection of deterministic failures. The former, as in the case of indexing, complements its effect with the program transformation. The latter aspect causes the shown problems.

Remark that infinite computations due to the interaction of the program transformation and dynamic cuts only occur, if the original program (without both optimizations) also runs into an infinite computation. Hence, in most practical applications, the undesired interaction does not occur. Nevertheless, for the sake of completeness, we still propose a variant of the program transformation for which we can ensure in all cases a better behaviour than dynamic cut alone: given a rule $f\ t_1 \ldots t_n := b\ \rightarrow\ e$, we generate the additional new rules

$$f_c\ t_1 \ldots t_n := \mathcal{E}(b, \mathit{true})\ \rightarrow \mathit{fail}, \qquad \text{if } c \notin P(b\ \rightarrow\ e) \cup \{\mathit{any}\}$$

With this transformation, we still give dynamic cut chances for detecting deterministic failure, but avoid the evaluation of something which is going to produce a failure. The price to pay is that the pruning effect of the transformation is drastically decreased in most cases, and that infinite computations while trying to evaluate b are no longer avoided.

3.5 A Scheme for Higher Order Functions

Unfortunately, it is hard to extend the presented approach to higher order functions. If a function f is given as an argument to another function, there is normally no information, which result f has to produce later on when it is used. In fact, it can be used several times to produce different results. Hence in the transformed program, f has to be replaced by f_{any}, which looses the result direction.

Another program transformation T_2 is less efficient for first order programs, but it is better extendible to higher order functions and partial applications, since the result direction is added as late as possible. Unfortunately, it is not applicable to general polymorphic programs, but the types of the arguments of higher order functions are restricted. Such an argument must not contain a function with a type variable as result. The reason is that applications with a constructed type as result and other applications are handled differently. Without the above restriction, both situations cannot be distinguished.

The transformation T_2 transforms each rule as follows:

$$\mathit{ruletrans}_2(f\ t_1 \ldots t_n := e) := \begin{cases} \{f\ t_1 \ldots t_n\ d_c := \mathcal{E}(e, c) \mid c \in P(e)\}, & \text{if the result of } f \text{ has a constructed type} \\ \{f\ t_1 \ldots t_n := e\}, & \text{otherwise} \end{cases}$$

where for each constructor c, we assume a new nullary constructor d_c. If c is nullary, c itself can be used instead of d_c. The $\mathcal{E}$ scheme is modified as follows:

$$\mathcal{E}(f\ e_1 \ldots e_m,\ c) := \begin{cases} f\ \mathcal{E}(e_1, c_1) \ldots \mathcal{E}(e_m, c_m)\ c', & \text{if the result of } f\ e_1 \ldots e_m \text{ has a constructed type} \\ f\ \mathcal{E}(e_1, c_1) \ldots \mathcal{E}(e_m, c_m), & \text{otherwise} \end{cases} \qquad m \geq 0$$

where every rule for f has $n \geq m$ arguments (note: partial applications are possible)

$$\text{and } c_i := \begin{cases} c'_i, & \text{if every rule } f\ t_1 \ldots t_n := e \text{ with } c \in P(e) \text{ has} \\ & c'_i \text{ as the topmost constructor of } t_i \qquad\qquad (1 \leq i \leq m) \\ any, & \text{otherwise} \end{cases}$$

$$\text{and } c' := \begin{cases} d_c, & \text{if } c \neq any \\ X, & \text{otherwise, where } X \text{ is a "new" variable} \end{cases}$$

Here, free variables may be inserted into the body of a rule. This violates the restriction on free variables in the body. The reason for this restriction is that Babel functions have to be functions in the mathematical sense, i.e. deterministic. The restricted use of variables here for result directed computing does not affect this property and hence it is valid.

The rest of the $\mathcal{E}$ scheme is not modified, but additionally we have to handle the application of an expression to other expressions. The following definition will only be used, if no previous definition applies.

$$\mathcal{E}(e\ e_1 \ldots e_m,\ c) := \begin{cases} \mathcal{E}(e, any)\ \mathcal{E}(e_1, any) \ldots \mathcal{E}(e_m, any)\ c', & \text{if the result of } (e\ e_1 \ldots e_m) \\ & \quad \text{has a constructed type} \\ \mathcal{E}(e, any)\ \mathcal{E}(e_1, any) \ldots \mathcal{E}(e_m, any), & \text{otherwise} \end{cases}$$

$$\text{where } c' := \begin{cases} d_c, & \text{if } c \neq any \\ X, & \text{otherwise, where } X \text{ is a "new" variable} \end{cases}$$

Several improvements of this scheme are possible. Here, we only sketch two of them. Rules producing a result which is never desired can be omitted. Moreover, the additional argument can be omitted, if the result of a function always starts with the same constructor and the function is never partially applied. Let us now have a look at an example:

original program:

fun check: $(\alpha \rightarrow bool) \rightarrow (list\ \alpha)$
$\quad \rightarrow bool.$
check P [ ] := *false*.
check P [X|L] :=
P X $\rightarrow$ *true* $\square$ *check* P L.

transformed and simplified program:

fun check: $(\alpha \rightarrow bool) \rightarrow (list\ \alpha) \rightarrow bool$
$\quad \rightarrow bool.$
check P [ ] *false* := *false*.
check P $[X \mid L]$ *false* :=
$\neg(P\ X\ false) \rightarrow check\ P\ L\ false.$
check P $[X \mid L]$ *true* :=
$P\ X\ Y \rightarrow true\ \square\ check\ P\ L\ true.$

For simplicity, we assume that partial applications of constructors and predefined operations are not allowed.

As for T_1 it also holds for T_2 that, if the original program terminates, then the transformed program also terminates with the same result, but not vice versa.

3.6 Relation to Prolog

In logic languages like Prolog, a function is simulated by a predicate where the "result" is added as an additional argument. If a non-variable pattern is specified for this argument, Prolog will automatically compute result directed. Unfortunately, there is no distinction between the guard and the body of a rule in Prolog. This implies that, in general, dynamic cuts due to deterministic computations cannot be used in Prolog.

On the other hand, it is possible to achieve result directed computing by simulating a Prolog program in Babel. This can be done by using the mentioned technique replacing n-ary functions by $(n+1)$-ary boolean functions (predicates) [KLMR90]. The right hand side of each rule is transformed into the guard of a new rule, which has the trivial body *true*. Using this approach, the additional argument in the goal would in general be bound during unification or while evaluating the guard. Hence, the computation is no longer deterministic and a dynamic cut cannot be introduced.

4 Experimental Results

It is easy to find programs where result directed computation does not gain anything. On the other hand, there are programs (as the *foo* example), where the transformation may prune an infinite computation. By replacing the second rule for *falsefct* in the example by

$$falsefct\ (suc\ X) := X \leq n \wedge falsefct\ X$$

(where $\leq$ is defined by the rules $0 \leq X := \mathit{true}$ and $(\mathit{suc}\ X) \leq (\mathit{suc}\ Y) := X \leq Y$) programs can be constructed, where the transformations gain arbitrarily much (depending on n).

An interesting question is, how much a transformation gains for "practical" programs. In order to get an impression, we have tried some example programs and compared the runtimes of the original and the transformed programs. The runtimes were measured using an implementation of Babel on a Sun 4 based on a stack-oriented narrowing machine [Lo91]. This intermediate code interpreter can run in eager and in lazy mode. We have tried the example programs with both evaluation strategies, eager and lazy narrowing.

Moreover, we have investigated the influence of other optimizations, like full indexing [Ha92] and dynamic cuts.

		eager narrowing			lazy narrowing		
		original program	using T_1	using T_2	original program	using T_1	using T_2
no optimi-zation	sec	12.78	4.78	5.48	16.49	7.25	8.76
	Stack (Bytes)	728356	683424	837432	691464	433208	558320
	Trail	1	1	1	11228	8022	8423
	choice points	128721	48922	50125	114686	49724	54135
with indexing	sec	8.68	3.36	3.78	7.10	6.16	7.15
	Stack (Bytes)	344	51496	53100	240720	235908	240720
	Trail	1	1	1	7220	7220	7220
	choice points	802	1203	1203	18446	18446	18446
with dynamic cut	sec	10.90	4.32	4.61	13.98	6.35	7.38
	Stack (Bytes)	368	320	336	472	472	476
	Trail	1	1	1	6817	4817	6420
	choice points	128721	48922	50125	114686	49724	54135
indexing +dyn. cut	sec	8.91	3.33	3.78	6.90	5.94	6.93
	Stack (Bytes)	344	296	300	452	448	452
	Trail	1	1	1	4817	4817	4817
	choice points	802	1203	1203	18446	18446	18446

Table 1: Statistics of the cousin example (repeated 400 times).

Table 1 shows the runtimes, the maximal sizes of the stack and trail and the overall number of generated choice points for the original and the transformed programs, with and without other optimizations. Obviously, the presented program transformations improve the runtimes considerably, even when other optimizations are used. In the cousin example, a lot of rules for the *male*, *disjoint*, and *member* functions can be excluded. Note that especially the *male* rules with result *false* cannot be discarded by indexing and dynamic cuts, but only by result directed computing. The advantage of the transformed programs is decreased by the fact that "$\wedge$" is handled by special code, while "$\odot$" is handled by predefined Babel rules. This causes the transformed programs to generate more choice points and require more stack space. If "$\odot$" were also handled by special code, the programs transformed by T_1 would (for all criteria) never be worse than the original program.

5 Conclusion

We have presented two program transformation schemes for result directed computing in a functional logic language, T_1 is more efficient for first order programs and T_2 is extendible to higher order functions. In contrast to a direct implementation of result directed computing by modifying the abstract machine, T_1 does not cause any runtime overhead. Unfortunately, both transformations may increase the size of the program. In the worst case, each rule may be replaced by $n+1$ rules (T_1) or n rules (T_2) respectively, where n is the number of constructors of the result type of the corresponding function. Since the number of constructors of a given type is typically small, this will be acceptable in most practical applications. On the other hand, as for other optimizations, it is perfectly valid, only to transform such expressions and rules, where the improvement is considerable compared to the increase of the program size. Both transformations may even be mixed. It is possible to use T_1 for functions, which have a constructed result type and which are never partially applied, and T_2 otherwise. Our approach can be generalized to take into account greater portions of the top of the result, although this is hardly worthwhile.

Acknowledgements

We thank Werner Hans and Stephan Winkler for helping us to perform the experimental results.

References

[Ai91] H. Aït-Kaci: Warrens's Abstract Machine: A Tutorial Reconstruction, MIT Press, 1991.

[BBCMMS89] G.P. Balboni, P.G. Bosco, C. Cecchi, R. Melen, C. Moiso, G. Sofi: Implementation of a Parallel Logic Plus Functional Language, in: P. Treleaven (ed.), *Parallel Computers: Object Oriented, Functional and Logic*, Wiley, 1989.

[BCGMP89] P.G. Bosco, C. Cecchi, E. Giovannetti, C. Moiso, C. Palamidessi: Using Resolution for a Sound and Efficient Integration of Logic and Functional Programming, in: J. de Bakker (ed.), *Languages for parallel architectures: Design, Semantics, Implementation Models*, Wiley, 1989.

[BEJ88] D. Bjørner, A.P. Ershov, N.D. Jones (eds.): Partial Evaluation and Mixed Computation, North Holland, 1988.

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

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

[Ha90] M. Hanus: Compiling Logic Programs with Equality, Workshop on Progr. Language Impl. and Logic Progr. (PLILP), LNCS 456, 1990, 387–401.

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

[HLW92] W. Hans, R. Loogen, S. Winkler: On the Interaction of Lazy Evaluation and Backtracking, to appear in: Procs. PLILP'92, LNCS, 1992.

[KLMR90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-based Implementation of a Functional Logic Language, Procs. ESOP'90, LNCS 432, 1990, 271-290.

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

[KLMR92b] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Implementing Disequality in a Lazy Functional Logic Language, to appear in: Procs. Symp. on Logic Programming'92, MIT Press, 1992.

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

[Lo91] R. Loogen: From Reduction Machines to Narrowing Machines, TAPSOFT'91, LNCS 494, 1991, 438-457.

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

[Mu90] A. Mück: Compilation of Narrowing, Workshop on Progr. Language Impl. and Logic Progr. (PLILP), LNCS 456, 1990, 16-39.

[MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, Procs. ALP'90, LNCS 463, 1990, 298-317; detailed version: Aachener Informatik-Bericht Nr. 90-11.

[MR88] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Babel: A functional and logic language based and constructor discipline and narrowing. Procs. ALP'89, LNCS 343, 1989, 223-232.

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

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

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

A The Full Transformation Scheme

$$\mathcal{E}(b \rightarrow e_1 \;\Box\; e_2,\ c) := \begin{cases} \mathcal{E}(b, any) \rightarrow \mathcal{E}(e_1, c) \;\Box\; \mathcal{E}(e_2, c), & \text{if } c \in P(e_1) \cap P(e_2) \\ \mathcal{E}(b, true) \rightarrow \mathcal{E}(e_1, c), & \text{if } c \in P(e_1) - P(e_2) \\ \neg\mathcal{E}(b, false) \rightarrow \mathcal{E}(e_2, c), & \text{if } c \in P(e_2) - P(e_1) \end{cases}$$

$$\mathcal{E}(b_1 \vee b_2,\ c) := \begin{cases} \mathcal{E}(b_1, true) \oplus \mathcal{E}(b_2, true), & \text{if } true \in P(b_1) \cap P(b_2),\ c = true \\ \mathcal{E}(b_1, true), & \text{if } true \in P(b_1) - P(b_2),\ c = true \\ \mathcal{E}(b_2, true), & \text{if } true \in P(b_2) - P(b_1),\ c = true \\ \mathcal{E}(b_1, c), & \text{if } c \in P(b_1) - P(b_2),\ c = false \\ \mathcal{E}(b_2, c), & \text{if } c \in P(b_2) - P(b_1),\ c = false \\ \mathcal{E}(b_1, c) \vee \mathcal{E}(b_2, c), & \text{if } (c \in P(b_1) \cap P(b_2) \text{ and } c = false) \text{ or } c = any \end{cases}$$

$$\mathcal{E}(b_1 \wedge b_2,\ c) := \begin{cases} \mathcal{E}(b_1, \mathit{false}) \odot \mathcal{E}(b_2, \mathit{false}), & \text{if } \mathit{false} \in P(b_1) \cap P(b_2),\ c = \mathit{false} \\ \mathcal{E}(b_1, \mathit{false}), & \text{if } \mathit{false} \in P(b_1) - P(b_2),\ c = \mathit{false} \\ \mathcal{E}(b_2, \mathit{false}), & \text{if } \mathit{false} \in P(b_2) - P(b_1),\ c = \mathit{false} \\ \mathcal{E}(b_1, c), & \text{if } c \in P(b_1) - P(b_2),\ c = \mathit{true} \\ \mathcal{E}(b_2, c), & \text{if } c \in P(b_2) - P(b_1),\ c = \mathit{true} \\ \mathcal{E}(b_1, c) \wedge \mathcal{E}(b_2, c), & \text{if } (c \in P(b_1) \cap P(b_2) \text{ and } c = \mathit{true}) \text{ or } c = \mathit{any} \end{cases}$$

The remaining definitions are unchanged w.r.t. the simplified scheme.