Abstract Interpretation of Functional Logic Languages

Werner Hans and Stephan Winkler*
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

January 12, 1993

Abstract

The technique of abstract interpretation is used in functional and logic programming with various applications. In this paper, we show how methods of abstract interpretation for logic programming can be adapted for purposes of functional logic languages. But these extensions are not straightforward — especially the presence of non-strict operations and a three-valued boolean domain cause additional problems. It is shown how the usage of slightly modified original techniques together with a program transformation that incorporates a *result directed analysis* can yield good results.

1 Introduction

During the last years, *functional logic languages* have been proposed in order to achieve the integration of functional and logic programming and their specific advantages. Usually, their syntax is influenced by functional programming paradigms, whereas for their operational semantics *narrowing* is used. Narrowing means a combination of the functional reduction mechanism with unification.

Abstract interpretation techniques with various applications have been studied for functional as well as logic programming [Cousot79, Abramsky86, Bruy91, Mellish81, Debray89, MutHerm89, JacLan89, Søndergaard86]. E.g., strictness analysis, occur check reduction, specialized unification, compile time garbage collection can be supported via static analysis. The incorporation of these techniques is also worthwhile for functional logic programming. Informally, abstract interpretation means the execution of the programs replacing the concrete domain by a simplified abstract one, which focuses on the interesting properties.

In the sequel we will concentrate on applications for the language BABEL [MorRod88]. Especially, the detection of pure functional computations may help to avoid overhead at runtime caused by the extension from reduction machines to narrowing machines [Loogen91]. For this purpose and in general, more optimized computations can be

*The authors are supported by the grant In 20/6–1 from the 'Deutsche Forschungsgemeinschaft'.
(e-mail address for correspondence: {hans, winkler}@informatik.rwth-aachen.de)

achieved with the knowledge about the argument modes of the functions. Additionally, mode analysis and independency analysis can be integrated in a parallelizing compiler.

The present paper is organized as follows. In Section 2 a short introduction to BABEL and its operational semantics are given. The next section focuses on abstract interpretation for logic programs with the subsequent extension for functional logic languages in Section 4. Particularly, the presence of non-strict operations causes additional problems that are discussed in detail. It is shown how slightly extended methods of logic programming together with a program transformation can gain safe results. Section 5 points out the lack of precision that might occur and describes an optimization adopted in the program transformation. Finally, some results show that a high degree of accuracy can be gained.

2 A Simple Functional Logic Language

In this section we define a first order subset of the functional logic language BABEL [MorRod88] which is based on a constructor discipline and which uses narrowing as evaluation mechanism.

2.1 Syntactic Domains

Let $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ be ranked alphabets of *constructors* and *function symbols*, respectively. The constructors 'true' and 'false', the boolean operations, and the equality operator are predefined. In the following, letters $c, d \ldots$ are used for constructors and the letters $f, g, \ldots$ for function symbols.

We distinguish between the following syntactic domains:

- *Variables* $X, Y, Z \ldots \in \mathit{Var}$

- *Terms* $s, t \ldots \in \mathit{Term}$:

$$
\begin{array}{lll}
t & ::= & X \qquad\qquad\quad \%\ \text{Variable} \\
 & \mid & c(t_1, \ldots, t_n) \quad \%\ c \in DC^n, n \geq 0
\end{array}
$$

- *Expressions* $M, N \ldots \in \mathit{Exp}$:

$$
\begin{array}{llll}
M & ::= & t & \%\ t \in \mathit{Term} \\
 & \mid & c(M_1, \ldots, M_n) & \%\ c \in DC^n, n \geq 0 \\
 & \mid & f(M_1, \ldots, M_n) & \%\ f \in FS^n, n \geq 0 \\
 & \mid & (B \rightarrow M) & \%\ \text{guarded expression} \\
 & \mid & (B \rightarrow M_1 \Box M_2) & \%\ \text{conditional expression}
\end{array}
$$

$B \rightarrow M$ and $B \rightarrow M_1 \Box M_2$ are intended to mean "**if** B **then** M **else** not defined" and "**if** B **then** M_1 **else** M_2", respectively.

2.2 Functional Logic Programs

A *BABEL program* consists of a finite set of defining rules for the not predefined function symbols in FS. Let $f \in FS^n$. Each *defining rule for* f must have the form:

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

and satisfy the following conditions:

1. *Data Pattern*: $t_i \in Term$.
2. *Left Linearity*: $f(t_1, \ldots, t_n)$ does not contain multiple variable occurrences.
3. *Restrictions on free variables*: Variables occurring in the right hand side but not in the left hand side are called *free*. Occurrences of free variables are allowed in the guard, but not in the body.
4. *Nonambiguity*: Given any two rules for the same function symbol f:
   $f(t_1, \ldots, t_n) := \{B \rightarrow\}M.$ and
   $f(s_1, \ldots, s_n) := \{C \rightarrow\}N.$
   one of the three following cases must hold:
   (a) *No superposition*: $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ are not unifiable.
   (b) *Fusion of bodies*: $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ have a most general unifier (m.g.u.) σ such that $M\sigma, N\sigma$ are identical. ($M\sigma$ denotes the expression M where all variables X have been replaced by $\sigma(X)$. $M\sigma$ is called an instance of M.)
   (c) *Incompatibility of guards*: $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ have a m.g.u. σ such that the boolean expressions $B\sigma$ and $C\sigma$ are incompatible.
   In [MorRod89] the notion of incompatibility has been formalized in such a way that
   (i) it is decidable whether two expressions B and B' are incompatible, and
   (ii) if B and B' are two incompatible first order expressions, there is no Herbrand algebra $\mathcal{A}$ such that, for some substitution ρ over $\mathcal{A}$:

$$[\![B]\!]_{\mathcal{A}}\rho = [\![B']\!]_{\mathcal{A}}\rho = \text{true}_{\mathcal{A}}$$

The formal definition of BABEL's declarative semantics is out of the scope of this paper. For our purpose it is only important, that the notion of incompatibility of expressions is chosen in such a way that incompatible expressions cannot simultaneously be evaluated to true.

Guarded rules allow a simple translation of Prolog clauses into BABEL. The body of the clause becomes the guard of the BABEL rule whose body is identical to true. To ensure left linearity the guard must be extended by appropriate equality conditions.

Note that the nonambiguity condition is trivially satisfied by logic programs, because the bodies of the rules are always identically true and thus condition (b) (fusion of bodies) applies.

If necessary, the following rules for the predefined function symbols are implicitly added to a BABEL program.

- Rules for the boolean operations:

not false	:=	true.
not true	:=	false.

false , Y	:=	false.
true , Y	:=	Y.

false ; Y	:=	Y.
true ; Y	:=	true.

- Rules for the equality operator '=':

$(c = c)$	:= true.	% $c \in DC^0$,
$(c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n))$	:= $(X_1 = Y_1), \ldots, (X_n = Y_n)$.	
		% $c \in DC^n$, $n > 0$
$(c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_m))$	:= false.	% $c \in DC^n$, $d \in DC^m$,
		% $c \neq d$ or $n \neq m$

2.3 Operational Semantics

The operational semantics of BABEL is based on the following *narrowing rule*, which describes how to apply a BABEL rule to an expression through unification:

> Let $f(t_1, \ldots, t_m) := R.$ be some variant of a rule in the program and $f(M_1, \ldots, M_m)$ an expression which shares no variables with the rule variant.
>
> If $\theta \cup \sigma : Var \rightarrow Exp$ is a most general unifier with $t_i\theta = M_i\sigma$ for $1 \leq i \leq m$, then we have:
> $$f(M_1, \ldots, M_m) \longrightarrow_\sigma R\theta.$$

To reduce guarded and conditional expressions one additionally needs the following special narrowing rules:

$(\text{true} \rightarrow M) \longrightarrow_\varepsilon M$ $\qquad$ $(\text{true} \rightarrow M_1 \Box M_2) \longrightarrow_\varepsilon M_1$

$(\text{false} \rightarrow M_1 \Box M_2) \longrightarrow_\varepsilon M_2$,

where ε denotes the empty substitution, i.e. $\varepsilon(X) = X$ for all $X \in Var$.

The *one step narrowing relation* $\Longrightarrow_\sigma \subseteq Exp \times Exp$ where $\sigma : Var \rightarrow Exp$ is inductively defined by:

- If $M_i \longrightarrow_\sigma N_i$ for $i \in \{1, \ldots, n\}$ then
  - $c(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma$ $c(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
  - $f(M_1, \ldots, M_i, \ldots, M_n) \Longrightarrow_\sigma$ $f(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)$
- $B \longrightarrow_\sigma B'$ implies

– $(B \rightarrow M) \Longrightarrow_{\sigma} (B' \rightarrow M\sigma)$ and $(B \rightarrow M_1 \Box M_2) \Longrightarrow_{\sigma} (B' \rightarrow M_1\sigma \Box M_2\sigma)$

The execution of several computation steps is given by the transitive, reflexive closure of the narrowing relation with the composition of the substitutions, $\stackrel{*}{\Longrightarrow}_{\sigma}$.
Narrowing of a BABEL expression M may lead to the following outcomes:

- *Success:* $M \stackrel{*}{\Longrightarrow}_{\sigma} t$ with $t \in Term$,
- *Failure:* $M \stackrel{*}{\Longrightarrow}_{\sigma} N$, N is not further narrowable and $N \notin Term$,
- *Nontermination.*

We consider in the following sections the *leftmost innermost* narrowing strategy.

3 Abstract Interpretation

Abstract interpretation is a well known technique for analyzing programs. Especially for logic programs, this technique can be applied in order to compute several properties of the program. The results can be used for optimization and parallelization purposes. For logic programs it is sufficient to compute all possible substitutions that can occur during runtime at some relevant program points. Since such a collection can be infinite, an analysis must deal with finite abstractions of the sets of substitutions. Therefore, an abstraction function α is given, which maps sets of concrete substitution to elements of an abstract domain $\mathcal{A}$. Usually, this domain is given by a lattice with an ordering relation $\sqsubseteq$. On the other hand, a concretization function γ maps an element of $\mathcal{A}$ to a concrete set of substitutions. Given a logic program and a (set of) queries, a safe analysis computes for each program point an abstract value λ that approximates all possible substitutions.

3.1 Abstract Interpretation of Logic Programs

In this approach we follow the top-down analysis of [Bruy91], which introduces AND-OR trees and operations for procedure entry, procedure exit and builtins. This interpretation mimics the concrete computation by generating an abstract AND-OR tree. An AND node corresponds to the evaluation of a right hand side whereas the OR nodes represent all clauses that can be applied for a given goal.

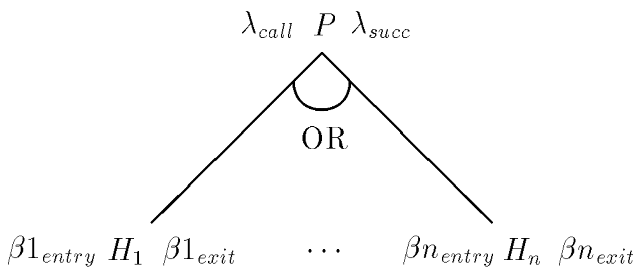


Figure 1: OR node in the abstract interpretation process

Consider the goal P with a *call substitution* λ_{call} (cf. Figure 1). Suppose that $H_1 \ldots H_n$ are the heads of clauses that unify with P. Abstract unification yields the *entry substitutions* $\beta 1_{entry} \ldots \beta n_{entry}$ from the call substitution. When the *exit substitutions* βi_{exit} of these clauses are available, the *success substitutions* λi_{succ} are evaluated from the exit substitutions and λ_{succ} by applying the LUB-operation (least upper bound) to all λi_{succ}.

Given a special entry substitution for a clause, the analyzer computes all intermediate substitutions λ_i (cf. Figure 2). λ_1 corresponds to the entry substitution and λ_{m+1} is the resulting exit substitution. These AND- and OR- trees are linked together, so that each SLD process is represented by a path in the AND-OR tree. If recursion occurs, this situation can be handled by the introduction of a backward edge and an iteration process, so that abstract AND-OR graphs are considered.

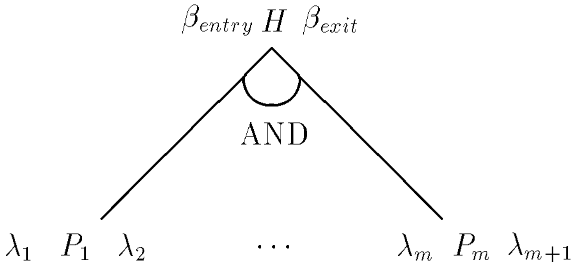


Figure 2: AND node in the abstract interpretation process

4 Abstract Interpretation of Functional Logic Languages

We give an extension of the framework of abstract interpretation to functional logic languages. We restrict ourselves in the first subsection on BABEL programs with only strict operations. Section 4 deals with the enlargement to non-strict operations.

4.1 Extension to Functional Logic Programs

The extension does not depend on the chosen abstract domain, except that we assume that the abstractions represent the respective sets of substitutions in the program points. For the sake of simplicity we assume that the execution of the program responds to an eager evaluation strategy and that the program does not possess any non-strict operation. So, the framework of Bruynooghe can be extended in the following way: Suppose that a BABEL rule

$$g\ (t_1, \ldots, t_n) := f_n(\ldots f_i(\ldots) \ldots)$$

is given. Since we handle consider strict operations and the eager strategy one can predict the order of the function calls of f_i in their chronological order during execution of the rule's right hand side. Therefore, the following AND node must be considered when applying the rule (cf. Figure 3).

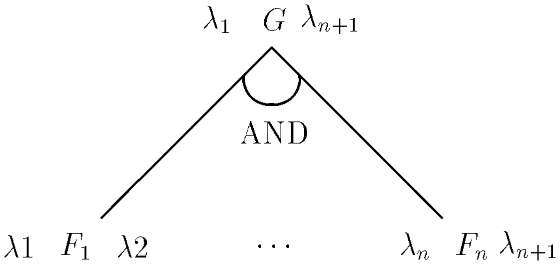


Figure 3: Execution of a right hand side

Similar to logic programs, the AND node visualizes the sequence of function calls. Nevertheless, the notion AND node is maintained, even if it has nothing in common with the boolean And-operation of BABEL. Clearly, the computed abstraction correspond to the program points before and after the function calls of the rhs, e.g.,

$$f(X) := g\ (h(X)).$$

$F_1 = h(X)$ and $F_2 = g(h(X))$. Since the result of the function call may appear as an argument of another one, the concerned operations in the interpretation process have to be extended. The operations are denoted by *function entry* and *function exit* and are defined as follows: *function entry* is defined equivalently to the procedure entry operation for logic programs, whereas the definition of *function exit* introduces some additional steps:

- perform the abstract unification of the function result of F_i with a new auxiliary variable X_i
- execute the procedure exit operation with respect to the extended abstract substitution of the previous unification
- replace each occurrence of F_i by X_i for the corresponding rule analysis

Equivalent to the logic case, alternative rules are managed by an OR node. According to these extensions, an AND-OR graph represents the narrowing processes in the concrete domain.

4.2 Interpretation of non-strict operations

When non-strict operations occur, the program analysis becomes more difficult, because it is in general not possible to determine whether all arguments of the operations must be evaluated or not. Note that the guarded expressions represents an exception because the abstract interpretation process is only interested in successful computations. But the analysis has to deal with all possibilities since the evaluations of the arguments may influence the instantiations of variables being accessed in some subsequent steps. Consider the abstract domain $\mathcal{A} = [V \to \{Var, Ground, Any\}]$ whose abstract values represent variables, ground terms, and any terms. We assume that an analysis should infer the modes of the BABEL program with the rule

$$f\ (X, Y) := (g(X), h(Y)).$$

containing the sequential And-operation (,) in the rhs. Since we consider a functional logic language with the three-valued boolean domain $\{\perp, false, true\}$, the computation may lead successfully to the result $false$ because of the function result of g, so that the invocation of h is suppressed. A safe analysis must deal with this possibility as well as the successful computation invoking both g and h.

This is necessary because the success substitutions of the rule differ in both cases when, for instance, h would bind the second argument Y with mode Var to a ground term. The general situation of a conjunction (g, h) can be expressed in a correct And-Or tree given in Figure 4.

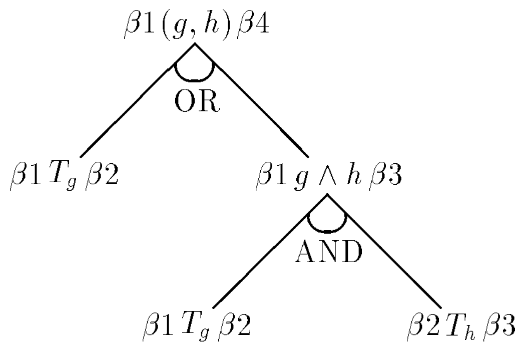


Figure 4: Conjunction of g and h

For both of the possibilities the corresponding subtrees are given. The first one invokes g whereas the second subtree shows the sequential computation of g and h. $\beta 4$ is computed by applying the LUB to $\beta 2$ and $\beta 3$.

Note that this problem is not interesting as far as a conjunction in a pure logic language is concerned. In this case the value $false$ would be interpreted as failure rather than a successful computation result. Consequently, a successful computation invokes both g and h.

This scheme is applied recursively to the subexpressions, so that AND-OR trees correspond to the evaluation of a rhs instead of a single AND node as for logic programs. In addition the sequential Or-operator and the If_then_else-operator should be considered. The Or-operation can be handled in the same way as above. Figure 5 shows the And-Or tree for If_then_else.

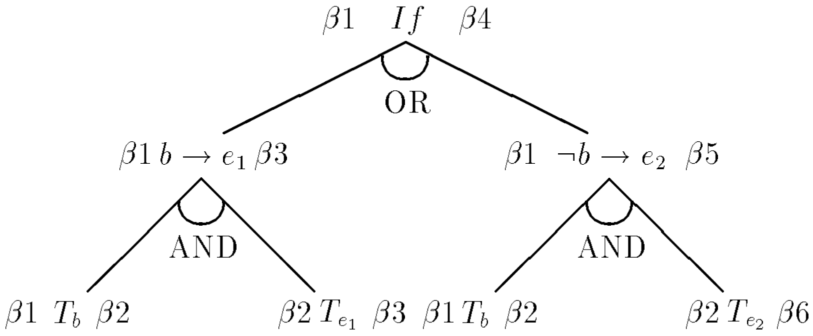


Figure 5: If_then_else-operation: $b \rightarrow e_1 \Box\, e_2$

Similarly, the AND-OR tree captures both branches of the concrete computation. In this case the LUB-operation below the OR node applied to β_3 and $\beta 5$ guarantees the safety of the analysis.

4.3 Implementation of a safe Analysis

The previous section shows the framework of a safe abstract interpretation of functional logic languages. Unfortunately, this approach introduces some additional kind of nodes

$$
\begin{array}{l}
\textbf{while}\ \textit{there exists a rule } r \textit{ with occurrence } e \textit{ of a non-strict operation}\\
\quad \textbf{begin}\\
\qquad \textit{let } x_1, \ldots x_n \textit{ the rule variables of } r \ ;\\
\qquad \textit{replace } e \textit{ by } f\ x_1, \ldots x_n \textit{ in } r\\
\qquad \textit{where } f \textit{ is a fresh function symbol} \ ;\\
\qquad \textbf{if}\ e = (e_1, (;)e_2) \textit{ add the rules}\\
\qquad\qquad f\ x_1, \ldots x_n := (e_1 \wedge (\vee) e_2).\\
\qquad\qquad f\ x_1, \ldots x_n := \ e_1.\\
\qquad\ \ \textit{to } P \textit{ with the strict operation } \wedge, \vee\\
\qquad \textbf{if}\ e = b \rightarrow e_1 \Box\ e_2 \textit{ add the rules}\\
\qquad\qquad f\ x_1, \ldots x_n := b \rightarrow e_1.\\
\qquad\qquad f\ x_1, \ldots x_n := \neg b \rightarrow e_2.\\
\qquad\ \ \textit{to } P\\
\quad \textbf{end}
\end{array}
$$

Figure 6: Program transformation

for non-strict operations so that existing algorithms have to be adjusted. On the other hand, program analysis is often realized by an interpreter, which mimics the concrete computation on the abstract domain. In such cases the introduced methods are hard to incorporate in the existing algorithm. Therefore, we will propose a program transformation algorithm that maps arbitrary functional logic programs to programs without non-strict operations to which the well known techniques and algorithms of logic languages can be applied. In addition, we show that the transformation is safe and conform to the method of the previous section.

Theorem 1: Abstract interpretation of functional logic programs with an innermost evaluation strategy and without non-strict operations can be handled by the extension of an analysis for logic languages as described in Figure 3. □

Proof Sketch:

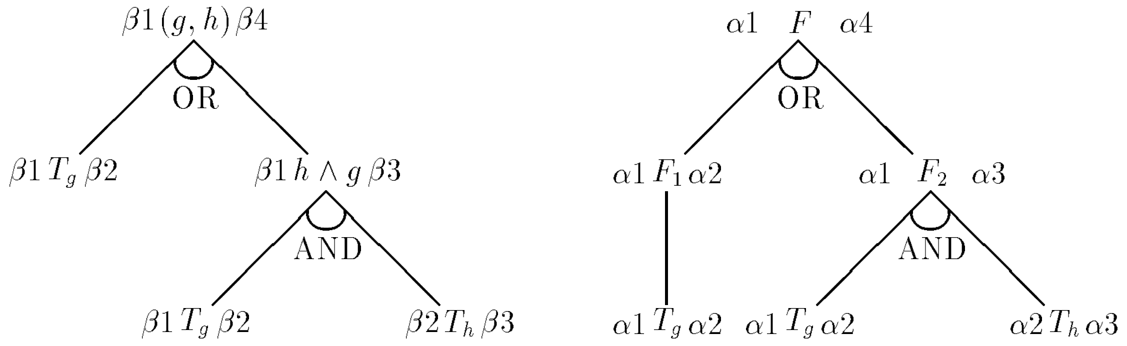
Since all expressions on the rhs have to be evaluated and the evaluation order is fixed, the function calls in Figure 3 correspond to the procedure calls in Figure 2, and the abstractions are inferred in the same way. Additionally, function results are treated by the introduction of auxiliary rule variables. With these modifications a correct abstract interpretation analysis would also yield correct results if function calls will be handled as procedure calls. □

As the theorem shows, the interpretation of programs on this simple class is no problem. The algorithm in Figure 6 shows the program transformation technique that captures correct results for an enlarged class with the operations for the sequential And,Or, and If_Then_Else:

The algorithm replaces every occurrence of non-strict functions by a new function call. Then the introduced functions capture all possibilities in several rules. The next theorem guarantees the correctness of the transformation.

Theorem 2: Applying abstract interpretation analysis to the transformed program yields safe (correct) results for all common programs points. Moreover, the computed abstractions are according to the programs analysis of Section 4.2. □

Proof: Since the introduced interpretation in Section 4.2 is safe, it is sufficient to show that the abstractions in common program points are equal. We will restrict ourselves to the points where the transformation is performed , e.g., the program points before and after the execution of a non-strict operation. First, we discuss the case of a conjunction (g, h), which is replaced by an application F with a fresh function.



On the upper picture the analysis of the original program is displayed. The picture below points out the situation when the analysis is executed on the transformed program. F_1 and F_2 are the corresponding lhs of the new rules. Apparently, the abstractions α_i and β_i are equal because of the equality of $\alpha1$ and $\beta1$ and the same structure of the tree. Only the first alternative of the new function causes the additional node (F_1), that can be neglected because the AND node has only one branch. The same situation is given for a disjunction whereas both AND-Or trees for If_then_else are identical. □

Note that the transformation is only applied for abstract interpretation purposes and that for the resulting programs the nonambiguity condition does not hold.

4.4 Specialization

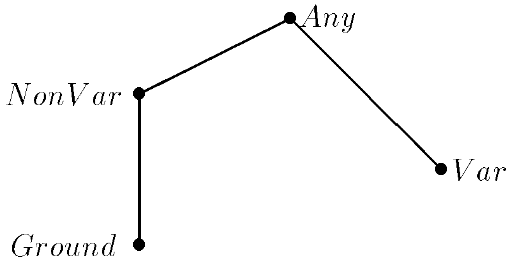
Up to now, we have considered only binary conjunctions. Let $(f_1, \ldots, f_n)$ a n-ary conjunction in a program rule. According to the previous description the safe analysis would consider each program point within the conjunction as a possibility to exit the operation. Therefore the transformation algorithm would generate at least n new rules. Fortunately, this is not necessary for each abstract domain. In this section we give an optimization and the conditions when these techniques can be applied.

Remark: Let $\mathcal{A}$ an abstract domain and $1, \ldots, n$ arbitrary program points within the body of a rule, which are ordered according to their evaluation order, and let α_i the corresponding abstract values. If $\bigsqcup_{i=1}^{n} \alpha_i = \alpha_1 \sqcup \alpha_n$ holds for all valid abstract values that might occur in the program points and for all rules then the transformation algorithm can be modified as follows:

- conjunctions (disjunctions) $(e_1, \ldots, e_n)$ are replaced by an application of a fresh function as before, but
- only two new rules
  $f\ x_1, \ldots x_n := (e_1 \wedge \ldots \wedge e_2).$
  $f\ x_1, \ldots x_n := e_1.$
  are added.

Clearly, the LUB-operation over all abstract values is superfluous in such cases.
Note that the former algorithm can deal this case as nested binary conjunctions or as a whole expression. But in each case at least n new rules would be generated so that a LUB-operation over all possibilities would affect the efficiency of the analysis.

Example 1: Consider the abstract domain $\mathcal{A} = [V \rightarrow \{Var, Any, NonVar, Ground\}]$ proposed in [Debray89]. Var denotes the set of non-instantiated variables, $Ground$ the set of ground terms, $NonVar$ the set of nonvariable terms, and Any the set of any terms. The set of abstract values forms a cpo under inclusion:



Now, it is obvious that the rule variables become more instantiated during rule evaluation following the instantiation order $Var \trianglelefteq Any \trianglelefteq NonVar \trianglelefteq Ground$. Thus, more than two different instantiations within a rule can yield only Var or Any at the beginning and $NonVar$ or $Ground$ at the end of the rule. In each case the LUB-operation of α_1 and α_n would yield Any so that the condition of the remark is fulfilled.

Unfortunately, the given condition of the remark does not hold when a sharing analysis with a corresponding domain is introduced.

Example 2: Consider the sharing domain over a set of program variables V as suggested in [JacLan89]. *Sharing* and *occurrence* are mappings $\mathcal{A} : Sub \rightarrow \mathcal{P}(\mathcal{P}(V))$ and $occ : Sub \times V \rightarrow \mathcal{P}(V)$ defined by

$$\mathcal{A}(\sigma) = \{occ(\sigma, x) \neq \emptyset \mid x \in V\}$$
$$occ(\sigma, x) = \{y \in V \mid x \in var(y\sigma)\}$$

While executing the right hand side as follows

$$l_1, X = Y, l_2, X = a, l_3$$

with the initial sharing $\{\{X\}, \{Y\}\}$, one can recognize that the sharing {{X,Y}} will be generated in the second literal and which disappears with the unification $X = a$.

5 Result Directed Analysis

We described the program transformation that enables the application of well known techniques for abstract interpretation with safe results. Unfortunately, the presence of some non-strict operations causes some inaccurate results that might be propagated to other parts of the program. In general, these results are not satisfactory for a program analysis. Thus, we have developed an improvement of the transformation, which allows us to infer precise results for most of our example programs. This improvement is based on a result oriented computation that is usually applied in concrete program computation as in [Kuchen,Lopez-Fraguas92]. This technique is refined and adjusted to our purposes and integrated in the transformation algorithm. The next subsections deal with an introduction to the technique and the results that can be gained by the optimization. As an example application we consider mode analysis for BABEL programs.

5.1 Transformation Scheme

As explained before the introduced transformation technique handles the various possibilities of evaluating the arguments of non-strict operations by introducing several rules. Then the LUB-operation of the corresponding exit substitution guarantees safety. On the other hand the success substitution can be affected by the different exit substitutions so that the overall result has a lack of precision. But in many cases it is possible to avoid the consideration of some alternatives because a special result is expected. This can be captured in an approach of *result directed analysis*, which is explained with the next example.

Example 3: Consider the queens program in BABEL, which is written in a Prolog-like style with the following rules

```
...
queens([],Ds,Es)        :=  Ds=Es  -> true.
queens(Dsunp,Dssafe,Ds) :=
       ( select(D, Dsunp, Ds1unp) ,
         noattack(D,Dssafe),
         queens(Ds1unp,[D|Dssafe],Ds) ) ->  true.
...
```

As one can recognize in the example above only the result *true* is useful as a result of the conjunction – otherwise a failure would occur while evaluating the guard. So the possible exit points within the conjunction after evaluating *select* and *noattack* can be neglected. Consequently, the corresponding rules generated by the transformation algorithm can be omitted. Notice that the expected result *true* might also affect the analysis of the three invoked functions of the conjunction because of other boolean connectives on their rhs. In the following, we present a more general scheme pointing out how the result directed analysis improves the transformation algorithm.

We start with the description of operations that incorporate the result direction within the rules. First *transform* generates for each rule three instances with the expected result of the function instance. The expected results in which we are mainly interested are *true* and *false*. All other cases are captured by *any*.

$$transform(f\ (t_1, \ldots, t_n := e.) := \{f_b\ (t_1, \ldots, t_n := \mathcal{E}(e, b). \mid b\ \in \{any, true, false\}\}$$

In a second step the scheme $\mathcal{E}$ is used in order to propagate the result direction correctly when invoking function and boolean operations on the rhs. Clearly, special instances of functions have to be invoked for this purpose and also some special treatments of the boolean operations are given.

$$\mathcal{E}(f(e_1, \ldots, e_n), b) := f_b\ (\mathcal{E}(e_1, any), \ldots, \mathcal{E}(e_n, any))$$

$$\mathcal{E}(c(e_1, \ldots, e_n), b) := c\ (\mathcal{E}(e_1, any), \ldots, \mathcal{E}(e_n, any))$$

$$\mathcal{E}(b' \to e, b) := \mathcal{E}(b', true) \to \mathcal{E}(e, b)$$

$$\mathcal{E}(b_1\ \wedge\ b_2, b) := \mathcal{E}(b_1, true)\ \wedge\ \mathcal{E}(b_2, b)$$

$$\mathcal{E}(b_1\ \vee\ b_2, b) := \mathcal{E}(b_1, false)\ \vee\ \mathcal{E}(b_2, b)$$

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

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

Now it is possible to specify the improved transformation algorithm as shown in Figure 7. The algorithm is based on the former transformation, so that two further steps are added. After applying *transform* to all rules the superfluous alternative rules for the concerned instances are omitted. Note that the scheme has to be applied to the given queries, too.

Theorem 3: The improved transformation algorithm applied to a BABEL program preserves safety for abstract interpretation. □

Proof: Clearly, the first three steps mean an extension to the former one with the incorporation of the expected result. Note that each other result would yield immediately failure because of some outer guard expression. What remains is the elimination of some alternatives for special rule instances as far as conjunctions and disjunctions are concerned. Since the omitted rules correspond to the mentioned failure situations, a further consideration of this would yield to additional abstractions of computation branches that never occur in the concrete narrowing process. Thus, these rules are superfluous and can be omitted without affecting safety. □

With these optimizations it is possible to infer all the modes of the queens programs for all functions and for all arguments exactly whereas without the optimization only one third of all are inferred exactly.

execute the program transformation of Figure 6;
apply $transform$ *to all rules of* P;
apply $\mathcal{E}$ *to all queries with initialization* any;

for all *functions* f *that are introduced by the first step*
  begin
    if f *replaced a conjunction*
      remove the second rule of f_{true};
    else if f *replaced a disjunction*
      remove the second rule of f_{false};
  end

Figure 7: Result oriented program transformation

5.2 Implementation Results

As an example application we give the results for mode analysis with and without the introduced optimization. The inferred modes can be used by a compiler in order to generate optimized code and for parallelization purposes. Originally, these techniques were proposed for logic programs [Mellish81] [Debray89], but there are similar applications in functional logic languages. Especially the detection of pure functional computations as in [Loogen,Winkler91] is worthwhile in BABEL so that run time checks can be avoided.

In general, the analysis shows good results as far as no non-strict operations are given. But problems occur when these operations together with logical variables are introduced. The comparison of the analysis behaviour for several programs of this kind is given in the table below. The table entries show the percentages of all exactly analyzed argument modes. The last column summarizes all results in relation to the total number of argument positions in the programs.

The comparison shows that there is a lack of accuracy without the optimization. Additionally, the inferred modes are worse for the success pattern. The situation changes when the optimization is used – one can recognize that the modes are inferred with a high degree of accuracy. What remains are only 7 % of argument modes that cannot be inferred exactly. This is due to the fact of some potential aliasing of arguments. But a combination together with an aliasing analysis as in [Hans,Winkler92] can achieve ratios of nearly 100 % when the optimization is used.

6 Conclusions

An extension of the abstract interpretation framework to functional logic programming has been presented. First, the function entry and exit operation were specified based on the operations for logic programs. We have introduced a program transformation that

program	call pattern		success pattern	
	simple	optimized	simple	optimized
queens	33	100	27	100
grammar	60	60	40	70
cousin	100	100	100	100
solitaire	47	100	53	100
puzzle	100	100	100	100
colours	100	100	62	100
palindrons	43	100	43	100
ackermann	60	60	60	60
relations	100	100	83	100
matrix	50	100	50	100
total	64	93	56	94

Table 1: Ratio of exactly inferred argument modes in percentage

eliminates the non-strict operation and in a second step the result directed transformation scheme has been embedded. Therefore, the methods of logic programs can be applied with the integrated program transformation technique. As an example application, we have considered mode analysis of BABEL programs. The examination of the results both of the original programs and its fully transformed versions have stressed that such an optimization is essential in order to achieve satisfactory results. The analysis yields a high degree of precision in practice even when non-strict operations occur.

Another interesting topic is the program analysis with respect to the lazy evaluation strategy. Then a combination with a strictness analysis seems to be promising. Future research will show whether the achieved accuracy can be kept.

References

[Abramsky86] S. Abramsky, Strictness Analysis and Polymorphic Invariance, *Proceedings of the DIKU Workshop on Programs as Data Objekts*, LNCS 217, 1985.

[Bruy91] M. Bruynooghe, A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming*, Vol. 10, 1991.

[Cousot79] P. Cousot, R. Cousot, Systematic Design of Program Analysis Framework, *6th POPL*, 1979.

[Debray89] S. K. Debray, Static Inference od Modes and Data Dependencies in Logic Programs, *North American Conference on Logic Programming*, MIT Press, 1989.

[Hans,Winkler92] W. Hans, St. Winkler, Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety, *Aachener Informatik Berichte 92–27*

[JacLan89] D. Jacobs and A. Langen, Accurate and Efficient Approximation of Variable Aliasing in Logic Programs, *North American Conference on Logic programming*, MIT Press, 1989

[Kuchen,Lopez-Fraguas92] H. Kuchen and F. J. Lopez-Fraguas, Result Oriented Computing in a Functional Logic Language, *Aachener Informatik Berichte 92–21*

[Loogen,Winkler91] R. Loogen and St. Winkler, Dynamic Detection of Determinism in Functional Logic Languages, Procs. PLILP'91, LNCS 528, 1991, 335-346

[Loogen91] R. Loogen: From Reduction Machines to Narrowing Machines, *Colloquium on Combining Paradigms for Software Development — CCPSD*, TAPSOFT 1991, LNCS, Springer Verlag 1991.

[MorRod88] J. J. Moreno-Navarro, M. Rodríguez-Artalejo: BABEL: A Functional and Logic Programming Language Based on Constructor Discipline and Narrowing, *Conf. on Algebraic and Logic Progr. 1988*, LNCS 343, Springer Verlag 1989, 223–232.

[MorRod89] J. J. Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language BABEL, *Technical Report DIA 89–3*, Universidad Complutense, Madrid, 1989.

[MutHerm89] K. Muthukumar, M. Hermenegildo, Determination of Variable Dependence Information Through Abstract Interpretation. *Proceedings of the North American Conference of Logic Programming*, Cleveland, Ohio, MIT Press, 1989.

[Mellish81] C. S. Mellish, The Automatic Generation of Mode Declaration for Prolog Programs *DAI Research Paper 163*, Dept. of Artificial Intelligence, Univ. of Edinburgh, Aug. 1981

[Søndergaard86] H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction, *Proceedings of the European Symposium on Programming*, LNCS 2123, 1986.