In *Proc. 1st International Static Analysis Symposium*, Namur (Belgium), pp. 26–42, Springer LNCS 864, 1994

Mode Analysis of Functional Logic Programs*

Michael Hanus Frank Zartmann

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany
`michael,frank@mpi-sb.mpg.de`

Abstract. Functional logic languages amalgamate functional and logic programming paradigms. They can be efficiently implemented by extending techniques known from logic programming. Such implementations can be largely improved if information about the run-time behavior, in particular the modes of function calls, is available at compile time. In this paper we present a framework to derive such global information. The concrete operational semantics considered in this paper is normalizing innermost narrowing, which combines the deterministic reduction principle of functional languages with the nondeterministic search principle of logic languages. Due to the normalization process between narrowing steps, standard analysis frameworks for logic programming cannot be applied. Therefore we develop new techniques to correctly approximate the effect of the intermediate normalization process.

1 Introduction

A lot of proposals have been made to amalgamate functional and logic programming languages (see [15] for a recent survey). Functional logic languages with a sound and complete operational semantics are based on narrowing (e.g., [9, 11, 25, 27]), a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing solves equations by finding appropriate values for variables occurring in goal equations. This is done by unifying an input term with the left-hand side of some rule and then replacing the instantiated input term by the instantiated right-hand side of the rule.

Example 1. The following rules define the addition of two natural numbers which are represented by terms built from `0` and `s`:

```
   0 + N → N                  (R1)
s(M) + N → s(M + N)           (R2)
```

To solve the equation `X+s(0)=s(s(0))`, we apply a narrowing step with rule R_2. This instantiates `X` to `s(M)`. The resulting left-hand side `s(M+s(0))` is narrowed with rule R_1 so that `M` is instantiated to `0`. Since the resulting equation, `s(s(0))=s(s(0))`, is trivially true, we have computed the solution `X↦s(0)` to the initial equation. □

* The research described in this paper was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103. The responsibility for the contents of this publication lies with the authors.

In order to ensure completeness in general, *each* rule must be unified with *each* non-variable subterm of the given equation which yields a huge search space. This situation can be improved by particular narrowing strategies which restrict the possible positions for the application of the next narrowing step (see [15] for a detailed survey). In this paper we are interested in an *innermost narrowing* strategy where a narrowing step is performed at the leftmost innermost position. This corresponds to eager evaluation in functional languages.

However, the restriction to particular narrowing positions is not sufficient to avoid a lot of useless derivations since the uncontrolled instantiation of variables may cause infinite loops. For instance, consider the rules in Example 1 and the equation `(X+Y)+Z=0`. Applying innermost narrowing to this equation using rule R_2 produces the following infinite derivation (the instantiation of variables occurring in the equation is recorded at the derivation arrow):

$$\texttt{(X+Y)+Z=0} \leadsto_{\texttt{X}\mapsto\texttt{s(X1)}} \texttt{s(X1+Y)+Z=0} \leadsto_{\texttt{X1}\mapsto\texttt{s(X2)}} \texttt{s(s(X2+Y))+Z=0} \leadsto_{\texttt{X2}\mapsto\texttt{s(X3)}} \cdots$$

To avoid such useless derivations, narrowing can be combined with simplification (evaluation of a term): Before a narrowing step is applied, the equation is rewritten to normal form w.r.t. the given rules [8, 9] (thus this strategy is also called *normalizing narrowing*). The infinite narrowing derivation above is avoided by rewriting the first derived equation to normal form:

$$\texttt{s(X1+Y)+Z=0} \;\rightarrow\; \texttt{s((X1+Y)+Z)=0}$$

The last equation can never be satisfied since the terms `s((X1+Y)+Z)` and `0` are always different due to the absence of rules for the symbols `s` and `0`. Hence we can safely terminate the unsuccessful narrowing derivation at this point.

Generally, the integration of rewriting into narrowing derivations yields a better control strategy than Prolog's SLD-resolution due to the reduction of the search space and the preference for deterministic computations (see [9, 12, 13] for more details).[2] Therefore we consider in this paper a *normalizing innermost narrowing* strategy where the computation of the normal form between narrowing steps is performed by applying rewrite rules from innermost to outermost positions, i.e., a rewrite rule is applied to a term only if each of its subterms is in normal form. Such an operational semantics can be efficiently implemented by extending compilation techniques known from logic programming [11, 12].

In logic programming it has been shown that the efficiency of programs can be largely improved if information about particular run-time properties is available at compile time (e.g., [22, 24, 28, 29, 30, 31, 32]). Moreover, in [16] it has been shown that there are useful optimizations which are unique to functional logic programs based on a normalizing narrowing strategy like ALF [11, 12], LPG [2], or SLOG [9]. Thus we need methods to derive the necessary information about the run-time behavior at compile time. The following example demonstrates that standard methods for the analysis of logic programs cannot be used.

[2] Note that the normalization of terms between narrowing steps is a deterministic process due to the uniqueness of normal forms.

Example 2. Consider the rules of Example 1 and the following additional rule:

`0*N` $\rightarrow$ `0` $\qquad (R_3)$

We are interested in the instantiation state of the variables after evaluating the goal `0*(X+Y)=Z`. From a logic programming point of view, where all subgoals are completely evaluated to prove the entire goal, we could infer that the evaluation of the innermost subterm `X+Y` binds `X` to a ground term before the outermost function `*` is evaluated. However, this is wrong if normalization is taken into account. Since the entire goal is normalized before a narrowing step is applied, the goal is reduced to `0=Z` by a rewrite step with rule R_3. Hence `X` remains unbound since the subterm `X+Y` is deleted during the normalization process. The deletion of subgoals has no correspondence in logic programming and therefore analysis methods for logic programming do not apply. □

This example shows that the analysis of normalizing narrowing requires a safe approximation of the effect of the normalization process before each narrowing step. After a precise definition of the operational semantics in Section 2, we review the notion of modes for functional logic programs in Section 3. We discuss problems related to the automatic derivation of modes in Section 4. In Section 5 we present our method to approximate modes at compile time. Due to lack of space, some details and the correctness proofs of the framework are omitted. They can be found in [33].

2 Normalizing Innermost Narrowing

In this section, we recall basic notions of term rewriting [7] in order to define the operational semantics considered in this paper.

A *signature* is a set $\mathcal{F}$ of *function symbols* together with their *arity*. If $\mathcal{X}$ is a countably infinite set of *variables* disjoint from $\mathcal{F}$, then $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of *terms* built from $\mathcal{F}$ and $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is called *ground* if $\mathcal{V}ar(t)=\emptyset$.

Usually, functional logic programs are *constructor-based*, i.e., a distinction is made between operation symbols to construct data terms, called *constructors*, and operation symbols to operate on data terms, called *defined functions* or *operations* (see, for instance, the functional logic languages ALF [11], BABEL [25], K-LEAF [10], SLOG [9]). Hence we assume that the signature $\mathcal{F}$ is partitioned into two sets $\mathcal{F}=\mathcal{C}\cup\mathcal{D}$ with $\mathcal{C}\cap\mathcal{D}=\emptyset$. A *constructor term* t is built from constructors and variables, i.e., $t\in\mathcal{T}(\mathcal{C},\mathcal{X})$. An *innermost term* t [9] is an operation applied to constructor terms, i.e., $t=f(t_1,\ldots,t_n)$ with $f\in\mathcal{D}$ and $t_1,\ldots,t_n\in\mathcal{T}(\mathcal{C},\mathcal{X})$. A *function call* $f(t_1,\ldots,t_n)$ is an operation $f\in\mathcal{D}$ applied to arbitrary terms.

A (*rewrite*) *rule* $l\rightarrow r$ is a pair of an innermost term l and a term r satisfying $\mathcal{V}ar(r)\subseteq\mathcal{V}ar(l)$ where l and r are called *left-hand side* and *right-hand side*, respectively. A rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. A *term rewriting system* $\mathcal{R}$ is a set of rules.[3] In the following we assume a given *term rewriting system* $\mathcal{R}$.

Substitutions and *most general unifiers* (*mgu*) are defined as usual. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [7] for details). $\mathcal{P}os(t)$ denotes the set of all positions in a term t and $\mathcal{NP}os(t)$ denotes the set of positions p of the term t with the property that $r|_p \in \mathcal{X}$ or $r|_p = f(\bar{s}),\ f \in \mathcal{D}$. The binary relation $<$ on $\mathcal{P}os(t)$ is the union of the relations $\{(p,q) \mid q \text{ is a proper prefix of } p\}$ and $\{(p,q) \mid p = \rho.i.p', q = \rho.j.q' \text{ and } i < j\}$. It reflects the leftmost innermost ordering.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{\mathcal{R}} s$ if there exist a position p in t, a rewrite rule $l \rightarrow r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. In this case we say t is *reducible*. A term t is called *irreducible* or in *normal form* if there is no term s with $t \rightarrow_{\mathcal{R}} s$.

$\rightarrow^*_{\mathcal{R}}$ denotes the transitive-reflexive closure of the rewrite relation $\rightarrow_{\mathcal{R}}$. $\mathcal{R}$ is called *terminating* if there are no infinite rewrite sequences $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \cdots$. $\mathcal{R}$ is called *confluent* if for all terms t, t_1, t_2 with $t \rightarrow^*_{\mathcal{R}} t_1$ and $t \rightarrow^*_{\mathcal{R}} t_2$ there exists a term t_3 with $t_1 \rightarrow^*_{\mathcal{R}} t_3$ and $t_2 \rightarrow^*_{\mathcal{R}} t_3$.

If $\mathcal{R}$ is confluent and terminating, we can decide the validity of an equation s=t by computing the normal form of both sides using an arbitrary sequence of rewrite steps. In order to *solve* an equation, we have to find appropriate instantiations for the variables in s and t. This can be done by *narrowing*. A term t is *narrowable* into a term t' if there exist a non-variable position p in t (i.e., $t|_p \notin \mathcal{X}$), a variant $l \rightarrow r$ of a rewrite rule and a substitution σ such that σ is a most general unifier of $t|_p$ and l and $t' = \sigma(t[r]_p)$. In this case we write $t \rightsquigarrow_\sigma t'$. In order to solve an equation s=t, we consider = as a new constructor symbol and apply narrowing steps until we obtain an equation s'=t' where s' and t' are unifiable. The composition of all unifiers in the derivation restricted to the variables of the initial equation is the *computed solution* (cf. Example 1). Since this simple narrowing procedure (enumerating all narrowing derivations) has a huge search space, several authors have improved it by restricting the admissible narrowing derivations (see [15] for a detailed survey). In the following we consider *normalizing innermost narrowing* derivations [9] where

- the narrowing step is performed at the leftmost innermost subterm, and
- the term is simplified to its normal form before a narrowing step is performed by applying rewrite rules from innermost to outermost positions.

The *innermost* strategy provides an efficient implementation [11, 12, 19, 21], whereas the *normalization* process is important since it prefers deterministic computations: rewriting a term to normal form can be done in a deterministic way

[3] We will apply rules in two ways: (a) in rewrite steps to evaluate terms, and (b) in narrowing steps to solve equations. Therefore we will sometimes distinguish between *rewrite rules* and *narrowing rules*. Usually, the set of rewrite rules and the set of narrowing rules are identical, but in some languages it is also possible to use some rules only for rewrite steps or only for narrowing steps (e.g., in ALF [11, 12] or SLOG [9]).

since every rewrite sequence yields the same result (because $\mathcal{R}$ is confluent and terminating), whereas different narrowing steps may lead to different solutions and therefore all admissible narrowing steps must be considered. Soundness and completeness results for this strategy can be found in [9].

3 Modes for Functional Logic Programs

It has been shown that mode information is useful to optimize the compiled code of pure logic programs [22, 24, 29, 31, 32]. A *mode* for a predicate is a description of the possible arguments of a predicate when it is called [32]. E.g., the mode $\mathtt{p}(g, f, a)$ specifies that the first argument is a *ground* term, the second argument is a *free* variable, and the third argument is an *arbitrary* term for all calls to predicate `p`. The notion of a "mode" in functional logic programs is different from pure logic programs because functions are evaluated by narrowing as well as by rewriting. In order to provide a better understanding of the subsequent sections, we review the notion of modes for functional logic programs as introduced in [16].

Example 3. In this example we discuss a derivation w.r.t. the normalizing innermost narrowing strategy. Consider the rules of Example 1 and the goal `X+(X+X)=s(s(s(0)))`. To compute a solution to this equation, we iterate the reduction to normal form with a subsequent narrowing step at the leftmost innermost subterm. Hence the left-hand side `X+(X+X)` is evaluated as follows (the rule applied in each step is listed in the rightmost column):

$$\begin{array}{lll} \mathtt{X+(X+X)} \leadsto_{\mathtt{X}\mapsto\mathtt{s(M)}} & \mathtt{s(M)+s(M+s(M))} & R_2 \\ \to_{\mathcal{R}} & \mathtt{s(M+s(M+s(M)))} & R_2 \\ \leadsto_{\mathtt{M}\mapsto\mathtt{0}} & \mathtt{s(0+s(s(0)))} & R_1 \\ \to_{\mathcal{R}} & \mathtt{s(s(s(0)))} & R_1 \end{array}$$

Since the term is already in normal form, the first step is a narrowing step at the inner subterm `X+X`. To normalize the resulting term, a rewrite step with rule R_2 is applied to the outermost occurrence of `+`. It follows a narrowing step at the inner subterm `M+s(M)` and a rewrite step at the remaining occurrence of `+`. Thus $\{\mathtt{X} \mapsto \mathtt{s(0)}\}$ is the computed solution. This derivation has the following interesting properties:

1. The operation `+` is evaluated both by narrowing and rewrite steps.
2. If a narrowing step is applied to `+`, the first argument is always an unbound variable.
3. If a rewrite step is applied to `+`, the first argument is partially instantiated.

□

Therefore we distinguish between a *narrowing mode* and a *rewrite mode* for each function. The narrowing mode describes the instantiation state of a function call if a narrowing step is applied to it ($\mathtt{+}(f, a)$ in the previous example) and the rewrite mode describes the instantiation state if a rewrite step is applied ($\mathtt{+}(a, a)$ in the previous example). Since narrowing and rewrite rules are usually compiled into different code sequences [11, 12], this distinction is necessary to optimize the compiled code, i.e., to specialize the unification/matching instructions and the indexing scheme (as done in pure logic programs). Moreover, using this kind of mode information it is possible to avoid unnecessary rewrite attempts, compile rewrite derivations in a more efficient way, delete unnecessary rewrite or narrowing rules etc. (see [16] for more details). However, a safe approximation of these modes is more complicated than in the pure logic programming case due to some global effects of the normalization process (cf. Example 2). In the following section we discuss these problems and potential solutions.

4 Automatic Derivation of Modes: Problems

Bosco et al. [3] have shown that innermost narrowing *without* normalization is equivalent to SLD-resolution if the functional logic program is transformed into a flat program without nested function calls. For instance, we could transform the rules of Examples 1 and 2 into the flat logic program

```
add(0,N,N).
add(s(M),N,s(Z)) :- add(M,N,Z).
mult(0,N,0).
```

where `add` and `mult` correspond to the functions `+` and `*` with their result values. The nested function call in the right-hand side of rule R_2 has been replaced by the new variable `Z` and the additional condition `add(M,N,Z)`. There is a strong correspondence between innermost narrowing derivations w.r.t. rules R_1, R_2 and R_3 and SLD-derivations w.r.t. the transformed logic program.

Due to these similarities of narrowing and SLD-resolution, one could try to apply abstract interpretation techniques developed for logic programming (e.g., [5, 20, 26]) to derive the desired information. E.g., to derive the narrowing mode of the function `+` w.r.t. the class of initial goals x+y=z, where x and y are always ground and z is a free variable, we could use an abstract interpretation framework for logic programming to infer the call modes of the predicate `add` w.r.t. the class of initial goals `add(`x`,`y`,`z`)`. In this case we infer that the call mode is `add`(g, g, f) and the argument z of the initial goal will be bound to a ground term at the end of a successful computation. Hence we could deduce that `+`(g, g) is the narrowing mode of the function `+`.

However, we have shown in Example 2 that normalizing innermost narrowing does not directly correspond to SLD-resolution because of the intermediate normalization process. For instance, the flat form of the equation `0*(X+Y)=Z` is the goal

```
add(X,Y,R), mult(0,R,Z).
```

The execution of the latter goal by SLD-resolution binds variable `X` to a ground term, whereas the execution of the original goal `0*(X+Y)=Z` by normalizing narrowing does not bind variable `X`. Therefore the analysis of the flattened logic program would yield an incorrect result.

This discussion shows that we cannot use a framework for the analysis of logic programs in our case. It is necessary to develop a new framework which takes into account the effect of normalization between narrowing steps. Since the accurate approximation of the normalization process is a challenging task, we will use the ideas of logic program analysis as long as possible, and we will introduce new analysis techniques only if it is unavoidable. This is a reasonable method since there are many functional logic programs where the "unpleasant" effects of normalization (from an analysis point of view) do not occur. Therefore we will distinguish between "pleasant" and "unpleasant" situations.

Different frameworks for the analysis of logic programs with a fixed left-to-right computation rule have been proposed in recent years (e.g., [5, 20, 26]). A common characteristic of these frameworks is the *locality of the analysis*: in order to derive information about the run-time behavior of the entire program, each clause is separately analyzed. The connection between the clauses and the goal literals activating the clauses is controlled by well-defined interfaces. For instance, from an analysis point of view a literal or predicate call L is considered as a function from call patterns into return patterns.[4] To compute or approximate this function, we take a clause $L_0 \leftarrow L_1, \ldots, L_n$, compute the mgu of L and L_0 and restrict the unifier to the variables occurring in this clause. The restricted unifier applied to L_1 yields the call pattern of the first literal in this clause and we proceed the analysis of the clause body where the return pattern of L_i is identical to the call pattern of L_{i+1} ($i = 1, \ldots, n-1$). The return pattern of the last literal L_n will be applied to L_0 and then unified with L. If we omit the information about the clause variables in this result, we obtain the result pattern of L. Since there is usually more than one applicable clause, we also analyze all other clauses in this way and compute the least upper bound of all result patterns.

Locality in this analysis means that during the analysis of the clause body $L_1, \ldots, L_n$ we do not consider the environment of L (i.e., the goal or clause body in which L occurs). This is justified since in a concrete computation the environment has no influence to the computation in the body. However, this is different in the case of functional logic programs due to the normalization process:

Example 4. Consider the following rules:

$$\begin{array}{rll} \texttt{f(c(a,Z))} & \rightarrow \texttt{a} & (R_1) \\ \texttt{g(X,Y)} & \rightarrow \texttt{c(h(X),h(Y))} & (R_2) \\ \texttt{h(a)} & \rightarrow \texttt{a} & (R_3) \end{array}$$

We want to compute the result pattern (here: modes) of the goal `f(g(X,Y))`. For this purpose, we analyze the right-hand side `c(h(X),h(Y))` of the rule for `g`. A local analysis would mean that we analyze the patterns for the function calls `h(X)` and `h(Y)`, and then infer the result pattern of the function call `g(X,Y)` (in this case: both arguments are bound to a ground term). However, we would obtain an incorrect result since the environment of this function call influence the evaluation of the right-hand side. This can be seen in the concrete derivation:

$$\texttt{f(g(X,Y))} \rightarrow_{\mathcal{R}} \texttt{f(c(h(X),h(Y)))} \rightsquigarrow_{\texttt{X}\mapsto\texttt{a}} \texttt{f(c(a,h(Y)))} \rightarrow_{\mathcal{R}} \texttt{a}$$

[4] A *pattern* is an abstract description of a set of concrete substitutions. For instance, the *mode pattern* $\texttt{add}(g,g,f)$ of a literal `add(X,Y,Z)` describes all substitutions which maps `X` and `Y` into ground terms and `Z` into a free variable.

Hence the variable `Y` remains free after the entire evaluation. Therefore we cannot analyze the rule for `g` without considering the environment. A more complex analysis method is necessary. □

Fortunately, this unpleasant case is rare and we often have the following situation: If s is a subterm of t, then the defined function symbols above s do not influence the evaluation of s, i.e., the ordering of narrowing steps inside s is not changed and s is completely evaluated before a narrowing step is applied outside s. Instead of giving a precise definition, we provide a sufficient and computable criterion to ensure that the context of s does not influence the evaluation of s. We say a subterm s at position p in t is *local* iff all defined function symbols above s preserve locality. The set of defined function symbols which *preserve locality* is the least set satisfying the following conditions. A defined function symbol preserves locality iff for all rules $f(\bar{u}) \to r$ for f, where $(X_1, \ldots, X_n)$ is the list of variables of $\bar{u}$ in leftmost innermost order, the following conditions are satisfied:

1. For all $j \in \{1, \ldots, n\}$ there is a position $p \in \mathcal{NP}os(r)$ with $r|_p = X_j$ and $\{r|_q \mid q \in \mathcal{NP}os(r), q < p\} = \{X_1, \ldots, X_{j-1}\}$.
2. All defined function symbols in r preserve locality.

The first condition demands that the rule does not delete subterms and ensures that the order of variables is preserved up to repetitions (this allows the rule `f(X,Y)`$\to$`c(X,X,Y,X)` but excludes `f(X,Y)`$\to$`c(Y,X)`). In the second condition we continue our demands on the defined functions in r. We denote by *LOC(r)* the set of positions of local subterms in a term r.

If a subterm is not local in a term, we have to take into account the effect of normalization during the analysis. Since the precise influence of normalization can only be approximated by the analysis, we obtain less accurate results in this case. In order to improve the accuracy of the analysis, we distinguish a class of subterms which allow a better analysis than in the general case. In many cases, functions with a nonlocal behavior on argument terms (like multiplication in Example 2) do not change the order of narrowing steps but simply deletes some possible narrowing steps (i.e., "possible" if normalization is not included). Since this allows a better analysis than in the general case, we want to characterize subterms s where the defined functions above s do not influence the ordering of narrowing steps in the derivation of s. Again, we provide a sufficient criterion for this property. We say a subterm s at position p in t is *weakly local* iff all defined function symbols above s preserve *weak locality.* The set of defined functions preserving weak locality is the least set satisfying the following conditions. A defined function symbol f *preserves weak locality* iff for all rules $f(\bar{u}) \to r$ for f, where $(X_1, \ldots, X_n)$ is the list of variables in $f(\bar{u})$ in innermost order, the following conditions are satisfied:

1. If $X_j \in var(r)$, then there exists $p \in \mathcal{NP}os(r)$ with $r|_p = X_j$ and $\{r|_q \mid q \in \mathcal{NP}os(r), q < p\} = \{X_1, \ldots, X_{j-1}\}$.
2. The defined function symbols in r preserve weak locality.

This definition is similar to the definition of defined function symbols preserving locality, but we do not require that all variables occurring in the left-hand side must also occur in the right-hand side. For instance, the function defined by `0*N`$\rightarrow$`0` preserves weak locality but not locality. We denote by *WLOC(r)* the set of positions of weakly local subterms in a term r. Note that $LOC(r) \subseteq WLOC(r)$.

The notions of locality and weak locality are sufficient to provide an accurate analysis for most practical programs. Therefore we give an overview of our analysis method in the next section.

5 Abstract Interpretation of Functional Logic Programs

Abstract interpretation is a systematic methodology to develop static program analysis methods [6]. The design of an abstract interpretation consists in defining an abstract domain AD which expresses relevant run-time information of programs. We assume that this abstract domain is a finite complete lattice.[5] Each element of an abstract domain represents a set of concrete elements, e.g., sets of substitutions. This relation is given by a concretization function γ. It maps an element of the abstract domain into the powerset of the concrete domain D. We assume that γ is an ordering morphism between the abstract domain and the powerset of the concrete domain endowed with the inclusion ordering: $\forall a, b \in AD:\ a \leq b \Rightarrow \gamma(a) \subseteq \gamma(b)$. The image of the bottom element $\perp \in AD$ should be the empty set and the image of the top element should be D. $a \sqcup b$ denotes the least upper bound of two elements $a, b \in AD$. We say that $a \in AD$ *approximates* $d \in D$, written $a \propto d$ iff $d \in \gamma(a)$. Further essential components of an abstract interpretation are operations on AD approximating the concrete operations on D. We assume familiarity with basic concepts of abstract interpretation.

5.1 Abstract Domains and Operations

We are interested in a general framework for the analysis of functional logic programs. Therefore we do not restrict ourselves to a particular abstract domain. We only assume that the abstract domain contains elements to describe substitutions over a fixed finite set V of variables. We denote the set of all these descriptions by AS_V. We abbreviate the abstract substitution best approximating the identity substitution by Id. In order to present examples for the analysis of modes in functional logic programs, we use in subsequent examples the product of the two domains $Mode_V$ and S_V, i.e., $AS_V = Mode_V \times S_V$. The first domain $Mode_V$ is a mapping of each variable in V into one of the four modes $g, f, a, \perp$. Each mode represents a set of constructor terms: $\gamma(\perp) = \emptyset$, $\gamma(a) = \mathcal{T}(\mathcal{C}, \mathcal{X})$, $\gamma(g) = \mathcal{T}(\mathcal{C}, \emptyset)$ and $\gamma(f) = \mathcal{X}$. The concretization function on $Mode_V$ is defined in the following way: $\sigma \in \gamma(\{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\})$ iff $\sigma(x_j) \in \gamma(m_j) \quad \forall j \in \{1, \ldots, n\}$. A correct analysis of freeness is not possible without considering the possible sharing

[5] It is possible to weaken this condition, but for the sake of simplicity we require a finite complete lattice.

between variables. Thus the second domain is the sharing domain $S_V = \mathcal{P}(\mathcal{P}(V))$ (sets of sets of variables from V) of Jacobs and Langen [17] with the following concretization function: $\sigma \in \gamma(S)$ iff for all $X \in \mathcal{V}ar(\sigma(v))$ for some $v \in V$ $\{y \mid X \in \mathcal{V}ar(\sigma(y))\} \subseteq A$ for some $A \in S$. We define γ on the entire domain by $\gamma((M,S)) = \gamma(M) \cap \gamma(S)$. Since $Mode_V$ and S_V are complete lattices, the product $Mode_V \times S_V$ is also a complete lattice.

In our analysis we have to approximate the evaluation of functions by normalizing narrowing. Thus we consider the derivation of $\sigma(r)$, where r is the right-hand side of a narrowing rule $R : f(\bar{u}) \to r$ or a part of a goal and σ is a constructor substitution. In Section 2 we have seen that the concrete computation is performed by applying narrowing steps with intermediate computations of the normal form, i.e., the concrete computation has the form

$$\sigma(r) \ \to^*_{\mathcal{R}} \ r_0 \ \leadsto_{\sigma_1} \ r_1 \ \to^*_{\mathcal{R}} \ r'_1 \ \leadsto_{\sigma_2} \ r_2 \ \to^*_{\mathcal{R}} \ r'_2 \cdots \ \leadsto_{\sigma_n} \ r_n \ \to^*_{\mathcal{R}} \ r'_n$$

As already discussed in Section 4, the potential problem in this derivation is the possibility that the normalization process changes the order of function calls. In particular, the leftmost innermost position in r'_i may be quite different from r_i. In order to obtain a correct approximation of such derivations, we will compute for each right-hand side r a sequence of states which approximates the sequence of narrowing steps in the derivation above. For this purpose we define the set of *computation states of a narrowing rule* $R : f(\bar{u}) \to r$ as

$$CS(R) = AS_V \times (WLOC(r) \cup \{\bot\})$$

where $V = \mathcal{V}ar(R)$. The first component A of a computation state $(A,p) \in CS(R)$ describes the instantiation of the rule variables, whereas the second component p describes the last narrowing position in r (or $\bot$ at the beginning of the derivation of r).[6]

In order to approximate the next narrowing position of the concrete computation, we have to analyze the behavior of the normalization process. For this purpose we use an extension of type graphs, a data structure introduced by Janssens and Bruynooghe [18] to describe sets of constructor terms. Our extended type graphs include additional information about the possible next narrowing position. We call these extended type graphs *term descriptions* and denote the set by $TD(R)$. Due to lack of space we cannot discuss the precise structure of term descriptions and the analysis of the normalization process (see [33] for more details). We only summarize those operations on $TD(R)$ which are necessary to understand the algorithm in Section 5.2.

The analysis of the normalization process is described as a family of functions

$$norm_R : AS_V \to TD(R)$$

[6] The sequence of computation states corresponds in some sense to the sequence of abstractions computed during the analysis of a clause body in abstract interpretation frameworks for logic programming [5]. However, the analysis of a clause body follows the left-to-right evaluation order of Prolog, whereas the sequence of computation states of a narrowing rule may not reflect the left-to-right innermost order in r since the normalization may restructure the subterms in r at run time.

(one for each narrowing rule $R : f(\bar{u}) \to r$ with $V = \mathcal{V}ar(R)$). A function $norm_R$ takes an approximation of the instantiation of the variables occurring in R and yields a description of the right-hand side after the normalization process. Due to the innermost normalization strategy, arguments are normalized before applying a rewrite rule to a function call. Thus the definition of $norm_R$ computes also the normalization of inner subterms which will be denoted by the function

$$norm\_args_R : AS_V \times \mathcal{P}os(r) \to AS_{V'} \times \mathcal{T}(\mathcal{C}, V')$$

The function $norm\_args_R$ takes a description of the rule variables and a position in the right-hand side and yields a description of the normalized arguments of the function call at this position. For instance, if the subterm `f(g(X),Y)` occurs in r at position 1 and the abstraction A implies that `X` is ground and `Y` free, then $norm\_args_R(A, 1) = (A', \texttt{(Z,Y)})$ where `Z` is a new variable representing the result of `g(X)` and A' implies that `Z` is ground (provided that the function `g` evaluates to a ground term if its argument is ground). Thus arguments containing defined functions are replaced by new variables describing the result of the argument evaluation, i.e., $V' \supseteq V$. In our analysis the new variables are only used to describe the effect of applying a narrowing rule at this position. Thus we omit these new variables after the rule application. For this purpose we need a *restrict function* $A|_V$ which maps an abstraction $A \in AS_{V'}$ into an abstraction $A' \in AS_V$ by forgetting the information about variables in $V' - V$.

The remaining auxiliary functions used in the analysis of a narrowing rule $R : f(\bar{u}) \to r$ with $V = \mathcal{V}ar(R)$ are summarized in the following table:

Auxiliary functions for the analysis of functional logic programs	
lub	$: \mathcal{P}(AS_V) \to AS_V$
$cons$	$: TD(R) \ \to Bool$
$func$	$: TD(R) \ \to Bool$
$leftmost$	$: TD(R) \ \to \mathcal{P}(WLOC(r) \cup \{(g(\bar{v}), A) \mid g \in \mathcal{D},\ A \in AS_{V \cup \mathcal{V}ar(\bar{v})}\})$

The function lub (*least upper bound*) takes a set of abstractions $\{A_1, \ldots, A_n\}$ and constructs a single abstraction which is the least upper bound $A_1 \sqcup \cdots \sqcup A_n$ of these abstractions. The predicate $cons(td)$ is satisfied if the denotation of a term description td may contain constructor terms. The predicate $func$ is satisfied if the denotation may contain a term with defined function symbols. The function $leftmost$ yields the set of possible function calls at the leftmost innermost position. Note that the leftmost innermost function cannot be uniquely determined in the analysis. Thus $leftmost$ returns a set of possible narrowing candidates. This set consists of weakly local subterms of r (i.e., function calls in the right-hand side of the rule) and new function calls possibly introduced during normalization in a subterm which is not weakly local. For instance, consider rule R_2 of Example 4. If the abstraction A describes `X` and `Y` as a (any possible term), the function $leftmost$ applied to $norm_{R_2}(A)$ yields the weakly local positions 1 and 2 of the subterms `h(X)` and `h(Y)`. This is a correct approximation since the concrete normalization depends on the exact instantiation of `X` and `Y`.

Finally, we use in our framework two operations $unify\text{-}aa : AS_V \times \mathcal{T}(\mathcal{C}, V) \times AS_W \times \mathcal{T}(\mathcal{C}, W) \to AS_W$ and $unify\text{-}ac : AS_V \times \mathcal{T}(\mathcal{C}, V) \times \mathcal{T}(\mathcal{C}, W) \to AS_V$

approximating unification.[7] If we assume that the abstract substitutions A and B describe substitutions σ and φ over disjoint sets V and W of variables, then $\mathit{unify\text{-}aa}(A, \bar{t}, B, \bar{s})$ describes the resulting substitution $mgu(\sigma(\bar{t}), \varphi(\bar{s})) \circ \varphi$. In contrast $\mathit{unify\text{-}ac}(A, \bar{t}, \bar{s})$ approximates $mgu(\sigma(\bar{t}), \bar{s}) \circ \sigma$, i.e., it describes the effect of unifying $\sigma(\bar{t})$ with $\bar{s}$ on the variables in $\bar{t}$.

5.2 The Analysis Algorithm for Functional Logic Programs

We want to analyze a rule $R : f(\bar{u}) \to r$.[8] As usual, the analysis of functional logic programs is a recursive process, and we describe the analysis as the least fixpoint of a system of recursive functions. For this purpose we define a *narrowing denotation* as a function

$$\delta : \mathit{Rules} \times AS_{\mathcal{X}} \times \mathcal{T}(\mathcal{C}, \mathcal{X}) \to AS_{\mathcal{X}} \times AS_{\mathcal{X}}$$

which maps a narrowing rule (Rules denotes the set of all narrowing rules in the program, where we assume that rules contain fresh variables if they are used in the analysis), a description and a list of current arguments into two other descriptions of the variables of the current arguments. Intuitively, if $\delta(f(\bar{u}) \to r, A, \bar{t}) = (A_1, A_2)$, then A_1 describes the possible instantiations of the variables in $\bar{t}$ *during* the derivation of the right-hand side r if this rule is applied to $f(\bar{t})$, whereas A_2 describes the instantiations *after* a successful derivation of $f(\bar{t})$. This distinction is necessary since it is sufficient to consider the complete result of the derivation only for local functions, whereas for weakly local functions it is also necessary to consider the intermediate states (since weakly local functions can be partially deleted, cf. Section 4).

If NDen denotes all narrowing denotations, we define our analysis as computing the least fixpoint of the operator $\Omega : \mathit{NDen} \to \mathit{NDen}$ with

$$\Omega(\delta)(R, A, \bar{t}) = \mathit{analyze}_R(\delta, A, \bar{t}) \ .$$

The family of functions $\mathit{analyze}_R$ approximates the behavior of a computation with narrowing rule R by generating all computation states for the right-hand side after the head unification (for convenience, we describe the functions of our algorithm with a free syntax, but it should be clear how to translate it into a pure functional language):

function $\mathit{analyze}_R(\delta \in \mathit{NDen}, A \in AS_{\mathcal{X}}, \bar{t} \in \mathcal{T}(\mathcal{C}, \mathcal{X})) : AS_{\mathcal{X}} \times AS_{\mathcal{X}}$
begin
$\psi_0 = (\mathit{unify\text{-}aa}(A, \bar{t}, Id, \bar{u}), \bot)$ % first state contains initial instantiations
$(\Upsilon, \Psi) = \mathit{generate\_states}_R(\delta, \psi_0)$ % generate subsequent states
return$(\mathit{back\_unify}_R(\Upsilon, \Psi, A, \bar{t}))$ % give the results back
end

The functions $\mathit{generate\_states}_R$ compute the transitive closure of the computation states of the right-hand side of each narrowing rule. Moreover, they collect

[7] For the sake of simplicity we consider an n-tuple of constructor terms also as a constructor term. This is always possible by introducing a pairing constructor symbol.
[8] W.l.o.g. we assume that the initial goal is also represented as a rewrite rule.

the intermediate patterns of local computations in the first component of the result. This is necessary to correctly approximate the intermediate states of surrounding weakly local functions (see $back\_unify_R$ below).

function $generate\_states_R(\delta \in NDen, \psi_0 \in CS(R)) : \mathcal{P}(AS_{\mathcal{X}}) \times \mathcal{P}(CS(R))$
 begin % $R = f(\bar{u}) \to r$
   $\Psi = \{\psi_0\}$; $\Upsilon = \emptyset$
   repeat % main loop: add new states to Ψ:
     for all $(A, \mu) \in \Psi$ **do**
       for all $l \in leftmost(norm_R(A))$ **do** % consider narrowing candidates
         if $l \in WLOC(r)$ **then** % candidate is weakly local function
           if $l \neq \mu$ **then** % candidate not considered before
             let $h(\bar{s}) = r|_l$ **and** $(A', \bar{t}) = norm\_args_R(A, l)$ **in**
             for all $h(\bar{w}) \to s \in Rules$ **do**
               $(I, F) = \delta(h(\bar{w}) \to s, A', \bar{t})|_{\mathcal{V}ar(R)}$ % analyze rules for h
               $\Psi = \Psi \cup \{(F, l)\}$ % add final state of the rule
               if $l \in LOC(r)$ **then** $\Upsilon = \Upsilon \cup \{I\}$
               else $\Psi = \Psi \cup \{(I, l)\}$ **fi** % add intermediate states
             od % for nonlocal functions
           fi
         else % global functions (i.e., not weakly local): compute effect of
           let $(h(\bar{t}), A') = l$ **in** % head unification with these functions
             $\Psi = \Psi \cup \{(unify\text{-}ac(A', \bar{t}, \bar{w})|_{\mathcal{V}ar(R)}, \perp) \mid h(\bar{w}) \to s \in Rules\}$
         fi
       od
     od
   until $\langle$no new states are added to $\Psi\rangle$
   return(Υ, Ψ)
 end

The functions $back\_unify_R$ compute upper bounds of all intermediate states and all final states of a narrowing rule:

function $back\_unify_R(\Upsilon \in \mathcal{P}(AS_{\mathcal{X}}), \Psi \in \mathcal{P}(CS(R)), A \in AS_{\mathcal{X}}, \bar{t} \in \mathcal{T}(\mathcal{C}, \mathcal{X})) :$
$AS_{\mathcal{X}} \times AS_{\mathcal{X}}$
 begin % $R = f(\bar{u}) \to r$
   $(I, F) = (\emptyset, \emptyset)$
   for all $(A', p) \in \Psi$ **do**
     $td = norm_R(A')$
     % add abstraction to I if rhs of R still contains function calls:
     if $func(td)$ **then** $I = I \cup \{unify\text{-}aa(A', \bar{u}, A, \bar{t})\}$ **fi**
     % add abstraction to F if the right-hand side of R is totally evaluated:
     if $cons(td)$ **then** $F = F \cup \{unify\text{-}aa(A', \bar{u}, A, \bar{t})\}$ **fi**
   od
   $I = I \cup \{unify\text{-}aa(B, \bar{u}, A, \bar{t}) \mid B \in \Upsilon\}$
   return$(lub(I), lub(F))$
 end

5.3 Derivation of Narrowing and Rewrite Modes

The main motivation of this work is the derivation of narrowing and rewrite modes for functional logic programs since they can be used to optimize the compiled programs in various ways (see Section 3 and [16]). The analysis presented so far does not derive these modes but approximates the instantiation of variables after a successful application of a narrowing rule. This is the most difficult task in the analysis due to the problems discussed in Section 4. Therefore it is easy to derive the narrowing and rewrite modes from our analysis. The narrowing modes can be inferred by collecting the initial modes of all narrowing rules computed in the functions $analyze_R$ and in $unify$-ac-calls. The computation of rewrite modes can be integrated in the functions $generate\_states_R$ and $norm_R$ by collecting all abstractions occurring during the normalization process.

5.4 Examples

In the following example we sketch the computed results of our algorithm. Since we are mainly interested in the mode component $Mode_V$ of the abstractions, we omit the sharing component in the example (although it is necessary to correctly derive freeness information).

Example 5. We discuss the analysis of a recursively defined function. We want to derive the modes of Example 3. For this purpose we represent the term of the initial goal as the right-hand side of a new rule:

$$
\begin{array}{rll@{\qquad}ll}
\texttt{0 + U} \rightarrow \texttt{U} & & (R_1) & \texttt{three(X)} \rightarrow \texttt{X+(X+X)} & (R_3) \\
\texttt{s(V) + W} \rightarrow \texttt{s(V + W)} & & (R_2) & &
\end{array}
$$

Since all functions are local, it is not necessary to consider the intermediate modes of narrowing rules. Hence we ignore these in the following discussion. The analysis starts by analyzing clause R_3 with the initial abstraction $A_0 = \{\texttt{X} \mapsto f\}$. Normalization of the right-hand side yields $leftmost(norm_{R_3}(A_0)) = \{2\}$, i.e., the subterm `X+X` is the next narrowing position. The analysis of rule R_1 is performed by $analyze_{R_1}(\bot, A_0, \texttt{(X,X)}) = (\ldots, \{\texttt{X} \mapsto g\})$ (note that we start a fixpoint computation with the undefined narrowing denotation $\bot$). To analyze the second rule, we compute the initial abstraction $A_1 = \{\texttt{V} \mapsto f, \texttt{W} \mapsto a\}$ of the right-hand side. The next narrowing position is the subterm `V+W`: $leftmost(norm_{R_2}(A_1)) = \{1\}$. The result of this recursive function call is approximated by the following chain of fixpoint iterations (note that we do not show the sharing component, but it is necessary to derive these results):

$$
\begin{array}{ll}
analyze_{R_1}(\bot, A_1, (V,W)) & = (\ldots, \{V \mapsto g, W \mapsto g\}) \\
analyze_{R_2}(\bot, A_1, (V,W)) & = (\ldots, \{V \mapsto \bot, W \mapsto \bot\}) \\
analyze_{R_1}(\Omega(\bot), A_1, (V,W)) & = (\ldots, \{V \mapsto g, W \mapsto g\}) \\
analyze_{R_2}(\Omega(\bot), A_1, (V,W)) & = (\ldots, \{V \mapsto g, W \mapsto g\})
\end{array}
$$

A further iteration does not change the narrowing denotations. Thus the analysis of R_2 w.r.t. A_0 yields the final result $\{\texttt{X} \mapsto g\}$, and we have reached a stable situation after one fixpoint iteration.

The narrowing modes for + can be derived by collecting all initial modes for the analysis of R_1 and R_2, i.e., $+(f, a)$ is the narrowing mode. Similarly, $+(a, a)$ is the derived rewrite mode. $\square$

Due to lack of space we cannot discuss the analysis of Example 4 in detail. Our analysis yields as success substitution for `f(g(X,Y))` the abstract value $\{\mathtt{X} \mapsto \mathtt{a}, \mathtt{Y} \mapsto \mathtt{a}\}$. The computed approximation is the worst possible one because of the presence of nonlocal function symbols. However, it is correct in contrast to an analysis of the corresponding flattened logic program.

6 Conclusions

In this paper we have presented a framework to approximate the run-time behavior of functional logic programs. The considered concrete operational semantics is normalizing narrowing, a combination of reduction, as used in pure functional languages, and nondeterministic narrowing steps, which are comparable to resolution steps in pure logic languages. This combination is very useful since it reduces the search space by the preference of deterministic evaluations and the deletion of complete subgoals during normalization. However, these useful effects makes an accurate approximation very difficult since the intermediate normalization process during narrowing steps may delete or restructure the order of subsequent narrowing steps. In order to catch this behavior at the abstract level, we have described the evolving computation of the right-hand side of each narrowing rule by a sequence of descriptions of the instantiation state of the rule variables. Depending on this instantiation state, the abstract normalization function yields a description of the next narrowing position in the right-hand side. To improve the accuracy of the analysis, we have distinguished two kinds of functions. *Local* functions cannot be deleted by the normalization of surrounding functions, and hence they are completely evaluated. Therefore it is only necessary to consider the success states of their corresponding narrowing rules. The order or narrowing steps in *weakly local* functions cannot be changed, but the evaluation may be cut off due to the deletion of arguments. Therefore they can be treated with a better accuracy than generally defined functions, but it is necessary to consider intermediate states in contrast to local functions.

The analysis of functional logic programs is a rather new research topic in the general area of program analysis. As far as we know, this paper is the first approach to derive mode information for functional logic languages based on normalizing narrowing. Boye [4] and Hanus [14] proposed methods to analyze functional logic programs where function calls are simply delayed until they are completely evaluable. Alpuente et al. [1] presented a framework to approximate the success patterns of terms evaluated by narrowing in order to detect unsolvable equations at analysis time. However, all these approaches do not cover the derivation of modes for function calls. As already discussed, this is a challenge in the presence of an operational semantics which dynamically restructure the call sequence in goals. Marriott et al. [23] proposed a framework to analyze logic programs where subgoals are dynamically delayed. This is in some sense related to the dynamic restructuring of goals, but it is different from our framework since we try to approximate also the complete deletion of subgoals which has no correspondence in logic programming. The possible deletion of subgoals is the main reason for the complexity of our approach.

The accuracy of our analysis depends on the locality of functions and the accuracy of the analysis of the normalization process. In our current framework, we use type graphs and mode information to approximate the normalization process and, in particular, the applicability of a rewrite rule. It is an interesting topic for future research to improve this approximation by stronger applicability conditions for rewrite rules using refined abstract domains. Another topic for future work is the implementation of this framework and the inclusion into an existing compiler in order to evaluate our analysis method on larger application programs.

References

1. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Inconsistency for Incremental Equational Logic Programming. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 443–457. Springer LNCS 631, 1992.
2. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. European Symposium on Programming*, pp. 119–132. Springer LNCS 213, 1986.
3. P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science 59*, pp. 3–23, 1988.
4. J. Boye. Avoiding Dynamic Delays in Functional Logic Programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 12–27. Springer LNCS 714, 1993.
5. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.
7. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
8. M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.
9. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
10. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
11. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.
12. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.
13. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.

14. M. Hanus. On the Completeness of Residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pp. 192–206. MIT Press, 1992.
15. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19-20, 1994.
16. M. Hanus. Towards the Global Optimization of Functional Logic Programs. In *Proc. 5th International Conference on Compiler Construction*, pp. 68–82. Springer LNCS 786, 1994.
17. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Proc. of the 1989 North American Conference on Logic Programming*, pp. 154–165. MIT Press, 1989.
18. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables. *Journal of Logic Programming*, Vol. 13, No. 2 & 3, pp. 205–258, 1992.
19. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. ESOP 90*, pp. 271–290. Springer LNCS 432, 1990.
20. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis. In *Proc. International Conference on Logic Programming*, pp. 64–78. MIT Press, 1991.
21. R. Loogen. Relating the Implementation Techniques of Functional and Functional Logic Languages. *New Generation Computing*, Vol. 11, pp. 179–215, 1993.
22. A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 33–47. MIT Press, 1989.
23. K. Marriott, M.J. Garcia de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 240–253, Portland, 1994.
24. C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.
25. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
26. U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 293–306. Springer LNCS 456, 1990.
27. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
28. A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 48–60. MIT Press, 1989.
29. A. Taylor. LIPS on a MIPS: Results form a Prolog Compiler for a RISC. In *Proc. Seventh International Conference on Logic Programming*, pp. 174–185. MIT Press, 1990.
30. P. Van Roy. An Intermediate Language to Support Prolog's Unification. In *Proc. of the 1989 North American Conference on Logic Programming*, pp. 1148–1164. MIT Press, 1989.
31. P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.
32. D.H.D. Warren. Implementing PROLOG - Compiling Logic Programs. 1 and 2. D.A.I. Research Report No. 39 and 40, University of Edinburgh, 1977.
33. F. Zartmann. Global Analysis of Functional Logic Programs. Technical Report, Max-Planck-Institut für Informatik, Saarbrücken, 1994.