In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming (Washington)*,
November 1992, pp. 192–206, MIT Press

On the Completeness of Residuation

Michael Hanus
Max-Planck-Institut für Informatik
Im Stadtwald, W-6600 Saarbrücken, Germany
michael@mpi-sb.mpg.de

Abstract

Residuation is an operational mechanism for the integration of functions into logic programming languages. The residuation principle delays the evaluation of functions during the unification process until the arguments are sufficiently instantiated. This has the advantage that the deterministic nature of functions is preserved but the disadvantage of incompleteness: if the variables in a delayed function call are not instantiated by the logic program, this function can never be evaluated and some answers which are logical consequences of the program are lost. In this paper we present a method for detecting such situations. The method is based on a compile-time analysis of the program and approximates the possible residuations and instantiation states of variables during program execution.

1 Introduction

Many proposals for the integration of functional and logic programming languages have been made during recent years (see [8] for a collection). From an operational point of view these proposals can be partitioned into two classes: approaches with a complete operational semantics and a nondeterministic search (*narrowing*) for solving equations with functional expressions (EQLOG [10], SLOG [9], K-LEAF [5], BABEL [16], ALF [11], among others), and approaches which try to avoid nondeterministic computations for functional expressions by reducing functional expressions only if the arguments are sufficiently instantiated (Funlog [20], Le Fun [3], LIFE [2], NUE-Prolog [17], among others). The former approaches are complete under some well-defined conditions (e.g., canonicity of the axioms), i.e., they compute all answers which can be logically inferred from the given program. The price for this completeness is an increased search space since there may be several incomparable unifiers of two terms if these terms contain unevaluated functional expressions. The latter approaches try to avoid this nondeterminism in the unification process. In these approaches a term is reduced to normal form before it is unified with another term, i.e., functional expressions are evaluated (if possible) before unification. If a function cannot be evaluated because the arguments are not sufficiently instantiated, the unification cannot proceed. Instead of causing a failure, the evaluation of the function is delayed until the arguments will be instantiated. This mechanism is called *residuation* in Le Fun [3]. For instance, consider the following program (we write residuating logic programs in the usual Prolog syntax but it is allowed to use arbitrary evaluable functions in terms):

```
q :- p(X,Y,5), pick(X,Y).
p(A,B,A+B).
pick(2,3).
```

together with the goal "`?- q`". After applying the first clause to the goal, the literals `p(X,Y,5)` and `p(A,B,A+B)` are unified. This binds `A` to `X` and `B` to `Y`, but the unification of `X+Y` and `5` is not successful since the arguments of the function call `X+Y` are not instantiated to numbers. Hence this unification causes the generation of the *residuation* `X+Y=5` which will be proved (or disproved) if `X` and `Y` will be bound to ground terms. We proceed by proving the literal `pick(X,Y)` which binds `X` and `Y` to `2` and `3`, respectively. As a consequence, the instantiated residuation `2+3=5` can be verified and therefore the entire goal has been proved.

The residuation principle seems to be preferable to the narrowing approaches since it preserves the deterministic nature of functions. However, it fails to compute all answers if functions are used in a logic programming manner. For instance, consider the function `append` for concatenating two lists. In a functional language with pattern-matching it can be defined by the following equations (we use the Prolog notation for lists):

```
append([],    L) = L
append([E|R],L) = [E|append(R,L)]
```

From a logic programming point of view we can compute the last element `E` of a given list `L` by solving the equation `append(_,[E])=L`. Since the first argument of the left-hand side of this equation will never be instantiated, residuation fails to compute the last element with this equation whereas narrowing computes the unique value for `E` [12]. Similarly, we can specify by the equation `append(LE,[_])=L` a list `LE` which is the result of deleting the last element in the list `L`. Combining the specification of the last element and the rest of a list, we define the reversing of a list by the following clauses:

```
rev([],[]).
rev(L, [E|LR]) :- append(LE,[E]) = L, rev(LE,LR).
```

Now consider the goal "`?- rev([a,b,c],R)`". Since the arguments of the calls to the function `append` are never instantiated to ground terms, the residuation principle cannot verify the corresponding residuation. Hence the answer `R=[c,b,a]` is not computed and there is an infinite derivation path using the residuation principle and applying the second clause infinitely many times. On the other hand a functional-logic language based on the narrowing principle can solve this goal and has a finite search space [12]. Therefore we should use narrowing instead of residuation in this example.

The last example raises the important question whether it is possible to detect the cases where the (more efficient) residuation principle is able to compute all answers. If this would be possible we can avoid the nondeterministic and hence expensive narrowing principle in many cases and replace it by computations based on the residuation principle without loosing any answers. A simple criterion to the completeness of residuation is the *ground-*

ness of all residuating variables: if at the end of a computation all variables occurring in residual function calls are bound to ground terms, then all residuations can be evaluated and hence the answer substitution does not depend on an unsolved residuation. Since the satisfaction of this criterion depends on the data flow during program execution, an exact answer is recursively undecidable. Therefore we present an approximation to this answer by applying abstract interpretation techniques to this kind of programs. Previous approaches for abstract interpretation of logic programs (see, for instance, [1, 7, 18]) depend on SLD-resolution as the operational semantics. Hence we cannot directly apply these frameworks to our case. However it is possible to develop a similar technique by considering unsolved residuations as part of the current substitution.

In the next section we give a short description of the operational semantics considered in this paper. The abstract domain and the abstract interpretation algorithm for reasoning about residuating programs are presented in Section 3. Finally, the correctness of our method is outlined in Section 4.

2 The residuation principle

In residuating logic programs terms are built from variables, constructors and (defined) functions. *Constructors* (denoted by `a`, `b`, `c`, `d`) are used to compose data structures, while defined *functions* (denoted by `f`, `g`, `h`) are operations on these data structures. We do not require any formalism for the specification of functions, i.e., they may be defined by equations or in a completely different language (external or predefined functions). However, the following conditions must be satisfied in order to reason about residuating logic programs:

1. A function call can be evaluated if all arguments are ground terms.
2. The result of the evaluation is a ground constructor term (containing only constructors) or an error message (i.e., the computation cannot proceed because of type errors, division by zero etc.).

The difference between residuating logic programs and ordinary logic programs shows up in the unification procedure: if a call to a defined function $f(t_1, \ldots, t_n)$ should be unified with a term t, the function call is evaluated if all arguments $t_1, \ldots, t_n$ are bound to ground terms and the unification proceeds with the evaluated term, otherwise the unification immediately succeeds and the *residuation* $f(t_1, \ldots, t_n) = t$ is added. If all variables in $t_1, \ldots, t_n$ will be bound to ground terms in the further computation process, the residuation $f(t_1, \ldots, t_n) = t$ will be immediately verified by evaluating the left-hand side and comparing the result with the right-hand side. Precise descriptions of this algorithm can be found in [3, 13] ([4] contains a more sophisticated version) and therefore we omit the details here. The result of the residuating unification algorithm is `fail` or a substitution/residuation pair $\langle \sigma, \rho \rangle$ with

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\} \qquad \text{and} \qquad \rho = \{s_1 = s'_1, \ldots, s_m = s'_m\}$$

where each variable x_i does not occur in t_j or ρ and s_i or s'_i are unevaluable (non-ground) function calls. In the entire computation σ is part of the answer substitution and ρ will be added to the unification problem in the next resolution step. The operational semantics of *residuating logic programs* considered in this paper is similar to Prolog's operational semantics (SLD-resolution with leftmost selection rule) but with the difference that the standard unification is replaced by a residuating unification algorithm. Thus the *concrete domain of computation* $\mathcal{C}$ is not simply the set of all substitutions but a set of substitution/residuation pairs, i.e.,

$$\mathcal{C} = \{\langle \sigma, \rho \rangle \mid \sigma \text{ is a substition, } \rho \text{ is a set of residuations}\}$$

where a residuation is an equation $r = r'$ and r (or r') is a function call. Since ground function calls are evaluated during unification, we assume in the following that all elements $\langle \sigma, \rho \rangle$ of the concrete domain $\mathcal{C}$ do not contain function calls with ground terms in the residuation part ρ.
As an example consider the following residuating logic program:

```
q :- p(X,Y,5), 1 = W-V, X = V*W, Y = V+W, pick(V,W).
p(A,B,A+B).
pick(1,2).
```

If the initial goal is `q`, the following elements of the concrete domain are computed during the processing of the first clause:

Before "`p(X,Y,5)`":	$\langle \emptyset,\ \emptyset \rangle$
After "`p(X,Y,5)`":	$\langle \emptyset,\ \{\texttt{5=X+Y}\} \rangle$
After "`1 = W-V`":	$\langle \emptyset,\ \{\texttt{5=X+Y, 1=W-V}\} \rangle$
After "`X = V*W`":	$\langle \{\texttt{X} \mapsto \texttt{V*W}\},\ \{\texttt{5=(V*W)+Y, 1=W-V}\} \rangle$
After "`Y = V+W`":	$\langle \{\texttt{X} \mapsto \texttt{V*W, Y} \mapsto \texttt{V+W}\},\ \{\texttt{5=(V*W)+(V+W), 1=W-V}\} \rangle$
After "`pick(V,W)`":	$\langle \{\texttt{X} \mapsto \texttt{1*2, Y} \mapsto \texttt{1+2, V} \mapsto \texttt{1, W} \mapsto \texttt{2}\},\ \emptyset \rangle$

At the clause end the residuation set is empty since all functions could be evaluated. Hence the initial goal is proved to be true.

From a semantical point of view residuations can be considered as *constraints* on substitutions and therefore the residuation framework could be viewed as a special case of the CLP framework [14]. However, this is not the case from an operational point of view. Since functions are user-defined, there need not exist a constraint solver which checks the satisfiablity of the accumulated residuations. E.g., the unsatisfiability of $\{$`append(L1,L2)=[1]`, `append(L2,L1)=[2]`$\}$ is not detected by the unification algorithms in [3, 4]. This would require a constraint solver for the defined list operations. In fact, it is reasonable to integrate the residuation principle into the CLP paradigm [19].

3 Abstract interpretation of residuating programs

In this section we present a method for checking whether the residuation part of the answer to a goal is empty, i.e., whether the residuation principle is complete w.r.t. a given program and goal. Since this problem is recursively undecidable in general, we present an approximation to it based on a compile-time analysis of the program. If this approximation yields a positive answer, then it is ensured that all residuations can be solved at run time. In the following we present the abstract domain and the motivation for it. The relation to the concrete domain and the correctness of the abstract interpretation algorithm are discussed in Section 4 in more detail. We assume familiarity with basic ideas of abstract interpretation techniques [1].

3.1 Abstract domain

There has been done a lot of work concerning the compile-time derivation of run-time properties of logic programs (see, for instance, the collection [1]). Since we have abstracted the different operational behaviour of residuating logic programs into an additional component of the concrete domain, we can use the well-known frameworks (e.g., [7, 18]) in a similar way. The heart of an abstract interpretation procedure is an abstract domain which approximates subsets of the concrete domain by finite representations. An element of the abstract domain describes common properties of a subset of the concrete domain. The properties must be chosen so that they contain relevant propositions about the interesting run-time properties. So what are the abstract properties in our case?

We are interested in unevaluated residuations at run time (second component of the concrete domain). A residuation can be verified if the function call in it can be evaluated. Since a function call can be evaluated if all arguments are ground, we need some information about the variables in it and the instantiation state of these variables in order to decide the emptiness of the residuation set. Hence our abstract domain contains information about the following properties:

Potential residuations: Residuations are generated by the unification of terms. For instance, if variable `X` is bound to `A+B` and variable `Y` is bound to `2` at run time, the unification of `X` and `Y` generates the residuation `A+B=2`. Hence, in order to state properties of all residuations which may occur at run time, we must know all potential function calls in the bindings of a program variable. Moreover, we must also know the variables in this function call in order to decide whether or not this function call can be evaluated. Therefore our abstract domain contains elements of the form "$\mathtt{X}\ \mathtt{with}\ \mathtt{+}|_{\{\mathtt{A},\mathtt{B}\}}$" meaning: variable `X` may be bound to a term containing a call to function `+` which can be evaluated if `A` and `B` are ground.

Dependencies between variables: Function calls can be evaluated if all variables in it are bound to ground terms. Hence we must have some information about the dependencies between variables. E.g., consider the goal

```
?- A+B = C,  C*2 = 6,  A = 1,  B = 2.
```

During unification of `C*2` and `6` the first term cannot be evaluated since `C` is not ground. But the groundness of `C` depends on the groundness of `A` and `B`.

Thus we can deduce that the function call `C*2` can be evaluated if `A` and `B` are bound to ground terms. Hence our abstract domain contains the element "$\mathtt{C}\ \mathtt{if}\ \{\mathtt{A},\mathtt{B}\}$". In general, "$\mathtt{X}\ \mathtt{if}\ V$" means that variable `X` is bound to a ground term if all variables in V are bound to ground terms.

Sharing between variables: The potential residuations can be copied between different variables in the unification process. E.g., consider the goal

```
?- Z = c(X),  Y = f(A),  X = Y, ...
```

After the unification of `X` and `Y` the variable `Z` contains the function call `f(A)`. In order to manage correctly the potential residuations, we must store the information that `Z` and `X` share a term. Hence our abstract domain contains the element $\{\mathtt{X},\mathtt{Z}\}$ representing the sharing between `X` and `Z`.

Summarizing the previous discussion, our *abstract domain* $\mathcal{A}$ contains the element $\perp$ (representing the empty subset of the concrete domain) and sets containing the following elements (such sets are called *abstractions* and denoted by A, A_1 etc):

Element:	Meaning:
$\mathtt{X}\ \mathtt{if}\ V$	`X` is ground if all variables in the variable set V are ground
$\mathtt{X}\ \mathtt{with}\ \mathtt{f}\vert_V$	`X` may be bound to a term containing a call to `f` which can be evaluated if all variables in V are ground
`f`	there may be an unevaluated function call to `f` depending on arbitrary variables
$\{\mathtt{X},\mathtt{Y}\}$	`X` and `Y` may share a term

Obviously, $\mathcal{A}$ is finite if the set of variables and function symbols is finite. Since we use only program variables and functions occurring in the program in the abstract domain, $\mathcal{A}$ is finite in case of a finite program. For convenience we simply write "`X`" instead of "$\mathtt{X}\ \mathtt{if}\ \emptyset$". Hence an element "`X`" in an abstraction means that variable `X` is bound to a ground term if it does not contain any function call.

Given an abstraction A, a variable X is called *function-free* in A if A does not contain elements of the form "$X\ \mathtt{with}\ f|_V$" and "f". In the subset of the concrete domain corresponding to A a function-free variable can only be interpreted as a term without unevaluable function calls (compare Section 4).

To present a simple description of the abstract interpretation algorithm, we will sometimes generate abstractions containing redundant information. The following *normalization rules* eliminate some redundancies in abstractions:

Normalization rules for abstractions:			
$A \cup \{Z,\ X\ \mathtt{if}\ V \cup \{Z\}\}$	$\to$	$A \cup \{Z,\ X\ \mathtt{if}\ V\}$	if Z is function-free in A
$A \cup \{Z,\ X\ \mathtt{with}\ f\vert_{V\cup\{Z\}}\}$	$\to$	$A \cup \{Z,\ X\ \mathtt{with}\ f\vert_V\}$	if Z is function-free in A
$A \cup \{X\ \mathtt{with}\ f\vert_\emptyset\}$	$\to$	A	
$A \cup \{X\ \mathtt{if}\ V_1,\ X\ \mathtt{if}\ V_2\}$	$\to$	$A \cup \{X\ \mathtt{if}\ V_1\}$	if $V_1 \subseteq V_2$
$A \cup \{X,\ \{X,Y\}\}$	$\to$	$A \cup \{X\}$	

The additional condition in the first two rules ensures that Z is bound to a ground term containing no unevaluable function calls. We call an abstraction A *normalized* if none of these normalization rules is applicable to A. Later we will see that the normalization rules are invariant w.r.t. the concrete substitutions/residuations corresponding to abstractions. Therefore we can assume that we *compute only with normalized abstractions* in the abstract interpretation algorithm.

In order to keep the abstract interpretation algorithm simple, we assume that predicate calls and clause heads have the form $p(X_1, \ldots, X_n)$ where all X_i are distinct (similarly to the example in [7]). All other literals in the clause bodies and goals have the form $X = Y$, $X = c(Y_1, \ldots, Y_n)$ or $X = f(Y_1, \ldots, Y_n)$. It is easy to see that every residuating logic program can be transformed into a *flat residuating logic program* satisfying the above restrictions without changing the answer behaviour. For instance, the residuating logic program in Section 2 can be transformed into the following equivalent flat program:

```
q :- Z=5,  p(X,Y,Z), T=1, T=W-V, X=V*W, Y=V+W, pick(V,W).
p(A,B,C) :- C=A+B.
pick(A,B) :- A=1, B=2.
```

In the following we assume that all programs are in the required form.

3.2 The abstract interpretation algorithm

The abstract interpretation algorithm is based on several operations on the abstract domain. The first operation restricts an abstraction A to a set of variables W. It will be used in a predicate call to omit the information about variables not passed from the predicate call to the applied clause:

$$
\begin{aligned}
\mathit{call\_restrict}(\perp, W) &= \perp \\
\mathit{call\_restrict}(A, W) &= \{X \in A \mid X \in W\} \\
&\quad \cup \{X \,\texttt{with}\, f|_V \in A \mid \{X\} \cup V \subseteq W\} \\
&\quad \cup \{f \mid f \in A \text{ or } X \,\texttt{with}\, f|_V \in A \text{ with } X \in W, V \not\subseteq W\} \\
&\quad \cup \{\{X,Y\} \in A \mid X, Y \in W\}
\end{aligned}
$$

The restriction operation for predicate calls transforms an abstraction element $X \,\texttt{with}\, f|_V$ into the element f if the dependent variables are not contained in W, i.e., it is noted that there may be an unevaluated function call to f but the possible dependencies are too complex for the abstract analysis. Similarly, an abstraction element of the form $X \,\texttt{if}\, V$ is passed to the clause only if $V = \emptyset$.

A similar operation is needed at the clause end to forget the abstract information about local clause variables. Hence we define:

$$
\begin{aligned}
\mathit{exit\_restrict}(\perp, W) &= \perp \\
\mathit{exit\_restrict}(A, W) &= \{X \,\texttt{if}\, V \in A \mid \{X\} \cup V \subseteq W\} \\
&\quad \cup \{X \,\texttt{with}\, f|_V \in A \mid \{X\} \cup V \subseteq W\} \\
&\quad \cup \{f \mid f \in A \text{ or } X \,\texttt{with}\, f|_V \in A \text{ with } \{X\} \cup V \not\subseteq W\} \\
&\quad \cup \{\{X,Y\} \in A \mid X, Y \in W\}
\end{aligned}
$$

The restriction operation for clause exits transforms an abstraction element $X \,\texttt{with}\, f|_V$ into the element f if one of the involved variables is not contained in W, i.e., it is noted that there may be an unevaluated function call to f which depends on local variables at the end of the clause.

The following operation computes the remaining abstract information of a predicate call restriction $call\_restrict(A, W)$ in order to combine it after a predicate call:

$$\begin{array}{rcl} rest(\bot, W) & = & \bot \\ rest(A, W) & = & \{X \texttt{ if } V \in A \mid X \notin W \text{ or } V \neq \emptyset\} \\ & & \cup \{X \texttt{ with } f|_V \in A \mid X \notin W\} \\ & & \cup \{\{X, Y\} \in A \mid X \notin W \text{ or } Y \notin W\} \end{array}$$

The *least upper bound* operation is used to combine the results of different clauses for a predicate call:

$$\begin{array}{rcl} \bot \sqcup A & = & A \\ A \sqcup \bot & = & A \\ A_1 \sqcup A_2 & = & \{X \texttt{ if } V_1 \cup V_2 \mid X \texttt{ if } V_1 \in A_1,\ X \texttt{ if } V_2 \in A_2\} \\ & & \cup \{X \texttt{ with } f|_V \mid X \texttt{ with } f|_V \in A_1 \text{ or } X \texttt{ with } f|_V \in A_2\} \\ & & \cup \{f \mid f \in A_1 \text{ or } f \in A_2\} \\ & & \cup \{\{X, Y\} \mid \{X, Y\} \in A_1 \text{ or } \{X, Y\} \in A_2\} \end{array}$$

Now we are able to define the abstract unification algorithm for the abstract interpretation of equations occurring in clause bodies or goals. Abstract unification is a function $au(\alpha, t_1, t_2)$ which takes an element of the abstract domain $\alpha \in \mathcal{A}$ and two terms t_1, t_2 as input and produces another abstract domain element as the result. Because of our restrictions on goal equations, the following definition is sufficient:[1]

$$\begin{array}{l} au(\bot, t_1, t_2) = \bot \\ au(A, X, X) = A \\ au(A, X, Y) = closure(A \cup \{X \texttt{ if } \{Y\},\ Y \texttt{ if } \{X\},\ \{X, Y\}\}) \quad \text{if } X \neq Y \\ au(A, X, c(Y_1, \ldots, Y_n)) = closure(A \cup \{X \texttt{ if } \{Y_1, \ldots, Y_n\},\ Y_1 \texttt{ if } \{X\}, \ldots, \\ \qquad\qquad Y_n \texttt{ if } \{X\},\ \{X, Y_1\}, \ldots,\ \{X, Y_n\}\}) \\ au(A, X, f(Y_1, \ldots, Y_n)) = closure(A \cup \{X \texttt{ if } \{Y_1, \ldots, Y_n\}, X \texttt{ with } f|_{\{Y_1, \ldots, Y_n\}}\}) \end{array}$$

In this definition and in the rest of this paper $closure(A)$ denotes the least set A' containing A which is closed under the following rules for transitivity and distribution of sharing information:

$$\begin{array}{rcl} \{X, Y\} \in A',\ \{Y, Z\} \in A' & \Longrightarrow & \{X, Z\} \in A' \\ \{X, Y\} \in A',\ X \texttt{ with } f|_V \in A' & \Longrightarrow & Y \texttt{ with } f|_V \in A' \end{array}$$

Now we can present the algorithm for the abstract interpretation of a residuating logic program in flat form. It is specified as a function $ai(\alpha, L)$ which takes an abstract domain element α and a goal literal L and yields a new abstract domain element as result. Clearly, $ai(\bot, L) = \bot$ and $ai(A, t = t') = au(A, t, t')$. The interesting case is the abstract interpretation of a predicate call $ai(A, p(X_1, \ldots, X_n))$ which is computed by the following steps:

[1]For simplicity we omit the occur check in the abstract unification.

1. Let $p(Z_1, \ldots, Z_n)$:- $L_1, \ldots, L_k$ be a clause for predicate p (if necessary, rename the clause variables such that they are disjoint from $X_1, \ldots, X_n$). Compute $A_{call} = call\_restrict(A, \{X_1, \ldots, X_n\})$
$$A_0 = \langle \text{replace all } X_i \text{ by } Z_i \text{ in } A_{call} \rangle$$
$$A_1 = ai(A_0, L_1);\ A_2 = ai(A_1, L_2);\ \ldots;\ A_k = ai(A_{k-1}, L_k)$$
$$A_{out} = exit\_restrict(A_k, \{Z_1, \ldots, Z_n\})$$
$$A_{exit} = \langle \text{replace all } Z_i \text{ by } X_i \text{ in } A_{out} \rangle$$
2. Let $A^1_{exit}, \ldots, A^m_{exit}$ be the exit substitutions of all clauses for p as computed in step 1. Then define $A_{success} = A^1_{exit} \sqcup \ldots \sqcup A^m_{exit}$
3. $ai(A, p(X_1, \ldots, X_n)) = closure(A_{success} \cup rest(A, \{X_1, \ldots, X_n\}))$ if $A_{success} \neq \perp$, else $\perp$

Hence a clause is interpreted in the following way. Firstly, the *call abstraction* is computed, i.e., the information contained in the predicate call abstraction is restricted to the argument variables (A_{call}). The variables in this call abstraction are mapped to the corresponding variables in the applied clause (A_0). Then each literal in the clause body is interpreted. The resulting abstraction (A_k) is restricted to the variables in the clause head, i.e., we forget the information about the local variables in the clause. Potential residuations which are unsolved at the clause end are passed to the abstraction A_{out} by the *exit_restrict* operation. In the last step the clause variables are renamed into the variables of the predicate call (A_{exit}). If all clauses defining the called predicate p are interpreted in this way, all possible interpretations are combined by the least upper bound of all abstractions ($A_{success}$). The combination of this abstraction with the information which was forgotten by the restriction at the beginning of the predicate call yields the abstraction after the predicate call (step 3).

The abstract interpretation algorithm described above is useless in case of recursive programs due to the nontermination of the algorithm. This classical problem is solved in all frameworks for abstract interpretation and therefore we do not want to develop a new solution to this problem but use one of the well-known solutions. Following Bruynooghe's framework [7] we can construct a rational abstract AND-OR-tree representing the computation of the abstract interpretation algorithm. During the construction of the tree we check before the interpretation of a predicate call P whether there is an ancestor node P' with a call to the same predicate and the same call abstraction (up to renaming of variables). If this is the case we take the success abstraction of P' (or $\perp$ if it is not available) as the success abstraction of P instead of interpreting P. If the further abstract interpretation computes a success abstraction A' for P' which differs from the success abstraction used for P, we start a recomputation beginning at P with A' as new success abstraction. This iteration terminates because all operations used in the abstract interpretation are monotone (w.r.t. the order on $\mathcal{A}$ defined in Section 4) and the abstract domain is finite.

3.3 An example

The following example is the flat form of a Le Fun program presented in [3]:

```
q(Z) :- p(X,Y,Z), X=V-W, Y=V+W, pick(V,W).
p(A,B,C) :- C=A*B.
pick(A,B) :- A=9, B=3.
```

The abstract interpretation algorithm computes the following abstractions w.r.t. the initial goal `q(T)` and the initial abstraction $\emptyset$ (specifying the set of all substitutions without unevaluated function calls):

$ai(\emptyset, \texttt{q(T)})$:

$ai(\emptyset, \texttt{p(X,Y,Z)})$: $ai(\emptyset, \texttt{C=A*B}) = \{\texttt{C}\,\texttt{if}\,\{\texttt{A,B}\},\ \texttt{C}\,\texttt{with}\,\texttt{*}|_{\{\texttt{A,B}\}}\}$

$ai(\emptyset, \texttt{p(X,Y,Z)}) = \{\texttt{Z}\,\texttt{if}\,\{\texttt{X,Y}\},\ \texttt{Z}\,\texttt{with}\,\texttt{*}|_{\{\texttt{X,Y}\}}\} =: A_1$

$ai(A_1, \texttt{X=V-W}) = \{\texttt{Z}\,\texttt{if}\,\{\texttt{X,Y}\}, \texttt{X}\,\texttt{if}\,\{\texttt{V,W}\}, \texttt{Z}\,\texttt{with}\,\texttt{*}|_{\{\texttt{X,Y}\}}, \texttt{X}\,\texttt{with}\,\texttt{-}|_{\{\texttt{V,W}\}}\} =: A_2$

$ai(A_2, \texttt{Y=V+W}) = \{\texttt{Z}\,\texttt{if}\,\{\texttt{X,Y}\},\ \texttt{X}\,\texttt{if}\,\{\texttt{V,W}\},\ \texttt{Y}\,\texttt{if}\,\{\texttt{V,W}\},$
$\qquad \texttt{Z}\,\texttt{with}\,\texttt{*}|_{\{\texttt{X,Y}\}},\ \texttt{X}\,\texttt{with}\,\texttt{-}|_{\{\texttt{V,W}\}},\ \texttt{Y}\,\texttt{with}\,\texttt{+}|_{\{\texttt{V,W}\}}\} =: A_3$

$ai(A_3, \texttt{pick(V,W)})$: $ai(\emptyset, \texttt{A=9}) = \{\texttt{A}\}$
$\qquad ai(\{\texttt{A}\}, \texttt{B=3}) = \{\texttt{A, B}\}$

$ai(A_3, \texttt{pick(V,W)}) = \{\texttt{V, W, Z}\,\texttt{if}\,\{\texttt{X,Y}\},\ \texttt{X}\,\texttt{if}\,\{\texttt{V,W}\},\ \texttt{Y}\,\texttt{if}\,\{\texttt{V,W}\},$
$\qquad \texttt{Z}\,\texttt{with}\,\texttt{*}|_{\{\texttt{X,Y}\}},\ \texttt{X}\,\texttt{with}\,\texttt{-}|_{\{\texttt{V,W}\}},\ \texttt{Y}\,\texttt{with}\,\texttt{+}|_{\{\texttt{V,W}\}}\}$
$\qquad \xrightarrow{normalize} \{\texttt{V, W, Z, X, Y}\}$

$ai(\emptyset, \texttt{q(T)}) = \{\texttt{T}\}$

Hence the computed success abstraction is $\{\texttt{T}\}$ meaning that after a successful computation of the goal `q(T)` the variable `T` is bound to a ground term and the residuation set is empty, i.e., the residuation principle allows to compute a fully evaluated answer. Similarly, the completeness of the residuation principle can be proved by our algorithm for all other residuating logic programs presented in [3]. A more complex example involving recursion can be found in [13].

4 Correctness of the abstract interpretation algorithm

In this section we will discuss the correctness of the presented abstract interpretation algorithm by relating the abstract domain to the concrete domain. Due to lack of space we omit the proofs of the theorems. The interested reader will find the proofs in [13].

To relate the computed abstract properties of the program to the concrete run-time behaviour, we have to define a *concretisation function* $\gamma{:}\mathcal{A} \to 2^{\mathcal{C}}$ which maps an abstraction into a subset of the concrete domain. The most difficult point in the definition of γ is the correct interpretation of an abstraction "$\texttt{X}\,\texttt{if}\,V$". The intuitive meaning is "the interpretation of `X` is ground if all interpretations of V are ground". To be more precise, "$\texttt{X}\,\texttt{if}\,V$" describes a dependency between the instantiation of $\mathtt{X}$ and the instantiation of the variables in V, i.e., we could define:

(*) If $\mathtt{X}\,\mathtt{if}\,V \in A$ and $\langle\sigma,\rho\rangle \in \gamma(A)$, then $var(\sigma(\mathtt{X})) \subseteq var(\sigma(V))$

($var(\xi)$ denotes the set of all variables occurring in the syntactic construction ξ). Such a definition seems to justify the generation of the abstractions "$\mathtt{X}\,\mathtt{if}\,\{\mathtt{Y}\}$" and "$\mathtt{Y}\,\mathtt{if}\,\{\mathtt{X}\}$" in the abstract unification algorithm if X is unified with Y. But this interpretation is not true if X or Y are bound to terms containing unevaluated residuations. E.g., if X is bound to f(B) and Y is bound to c(A) during program execution, then the computation of the literal X=Y yields the substitution/residuation pair $\langle\emptyset,\{\mathtt{f(B)=c(A)}\}\rangle$. Thus the variables contained in the bindings of X and Y are not identical after the unification step. Therefore we must weaken (*) to the condition that only the variables of $\sigma(\mathtt{X})$ occurring outside function calls are contained in the variables of $\sigma(V)$ *w.r.t. to the residuation* ρ.

To give a precise description of the condition, we need the following definitions. By $lvar(t)$ we denote the set of all variables occurring outside function calls in the term t:

$$\begin{array}{rcl} lvar(X) & = & \{X\} \\ lvar(c(t_1,\ldots,t_n)) & = & lvar(t_1) \cup \cdots \cup lvar(t_n) \\ lvar(f(t_1,\ldots,t_n)) & = & \emptyset \end{array}$$

The *extension* of a set of variables V *w.r.t. to the residuation* ρ is defined by

$$var_\rho(V) = V \cup \{lvar(e) \mid f(\overline{t}) = e \in \rho \text{ or } e = f(\overline{t}) \in \rho \text{ with } var(\overline{t}) \subseteq V\}$$

(where $\overline{t}$ denotes the argument sequence $t_1,\ldots,t_n$). Note that $var_\rho(\emptyset) = \emptyset$ if ρ does not contain unevaluated ground residual function calls (which do not occur in our concrete domain) and for an empty residuation we have $var_\emptyset(V) = V$. The intuition of this definition is that we add to a set of variables V all these variables which will be ground during the computation process if all variables in V are ground. For instance, if $\rho = \{\mathtt{f(X)=c(Y)},\mathtt{f(X)=c(Z)}\}$, then $var_\rho(\{\mathtt{X}\}) = \{\mathtt{X},\mathtt{Y},\mathtt{Z}\}$. We extend the function var_ρ to finite sets of terms by

$$var_\rho(\{t_1,\ldots,t_k\}) = var_\rho(var(\{t_1,\ldots,t_k\}))$$

Since we are interested in the property whether a function call occurring in a term can be completely evaluated, it is sufficient to look at the main function calls and not at function calls which occur inside other function calls (this is due to the fact that a unification between a function call and another term does not bind any variables in this call). Therefore we say a term t occurs *directly* in a term t' if t occurs in t' outside a function call. For instance, the term $X + (Y * 2)$ occurs directly in the term $c(X + (Y * 2))$ but the subterm $(Y * 2)$ is not a direct occurrence.

Now we are able to define the semantics of abstractions by the concretisation function $\gamma{:}\mathcal{A} \to 2^{\mathcal{C}}$ (where $\overline{t}$ denotes the argument sequence $t_1,\ldots,t_n$):

$$
\begin{array}{ll}
\gamma(\bot) = \emptyset & \\
\gamma(A) = \{\langle\sigma,\rho\rangle \in \mathcal{C} \mid & 1.\ X\,\texttt{if}\,V \in A \Rightarrow lvar(\sigma(X)) \subseteq var_\rho(\sigma(V)) \\
& 2.\ f(\bar{t}) \text{ occurs directly in } \sigma(X) \text{ or } \rho \text{ with } var(\bar{t}) \neq \emptyset \\
& \quad \Rightarrow f \in A \ \text{ or } \ var(\bar{t}) \subseteq var(\sigma(V)) \text{ for some } X\,\texttt{with}\,f|_V \in A \\
& 3.\ lvar(\sigma(X)) \cap lvar(\sigma(Y)) \neq \emptyset \text{ for } X \neq Y \Rightarrow \{X,Y\} \in A \quad \}
\end{array}
$$

Condition 1 implies for $X\,\texttt{if}\,V \in A$ that all variables occurring outside function calls in the current instantiation of X are ground if all variables in V are instantiated to ground terms. Condition 2 ensures that all unevaluated function calls in variable bindings and in residuations are contained in A. Since we are interested in potential residuations, it is sufficient to look at function calls which occur *directly* in some variable binding (and not at function calls nested in other function calls). Hence the sharing information is also restricted to $lvar$ instead of var (condition 3). Note that for an unevaluated function call in the residuation part it is sufficient that there is an arbitrary variable X which cover this function call whereas for an unevaluated function call in the binding of a variable X there must be an abstraction element $X\,\texttt{with}\,f|_V$ with the *same* variable. This is necessary for passing the correct information about potential residuations in case of a predicate call (compare call restriction operation).

From this interpretation it is clear that an abstraction without elements of the form "$X\,\texttt{with}\,f|_V$" or "f" can only be interpreted as a fully evaluated pair $\langle\sigma,\rho\rangle$ if $\rho = \emptyset$ and σ does not contain unevaluable function calls. This argument has been used to state the completeness of the example in Section 3.3.

Due to this semantics of abstractions it can be proved that the normalization rules defined on abstractions in Section 3.1 are invariant w.r.t. the concrete interpretation. The following lemma justifies the application of the normalization rules.

Lemma 4.1 *If A and A' are abstractions with $A \rightarrow A'$, then $\gamma(A) = \gamma(A')$.*

For the termination of the abstract interpretation algorithm it is important that all operations on the abstract domain are monotone. Therefore we define the following order relation on normalized abstractions:

$$
\begin{array}{lll}
\text{(a)} & \bot \sqsubseteq \alpha \text{ for all } \alpha \in \mathcal{A} & \\
\text{(b)} & A \sqsubseteq A' \iff & 1.\ X\,\texttt{if}\,V' \in A' \ \Rightarrow\ \exists V \subseteq V' \text{ with } X\,\texttt{if}\,V \in A \\
& & 2.\ X\,\texttt{with}\,f|_V \in A \ \Rightarrow\ X\,\texttt{with}\,f|_V \in A' \\
& & 3.\ f \in A \ \Rightarrow\ f \in A' \\
& & 4.\ \{X,Y\} \in A \ \Rightarrow\ \{X,Y\} \in A'
\end{array}
$$

It is easy to prove that $\sqsubseteq$ is a reflexive, transitive and anti-symmetric relation on normalized abstractions, the operation $\sqcup$ defined in Section 3.2 computes the least upper bound of two abstractions, and γ is monotone.

The correctness of the abstract interpretation algorithm is based on the correctness of each component of the algorithm. The entire proof can be constructed following the ideas in [7]. Due to the complex abstract domain the detailed proofs require some effort and cannot be shown in this paper. In the following we only state an important theorem which is the basis for the correctness of the abstract interpretation algorithm:

Theorem 4.2 (Correctness of abstract unification) *Let X be a variable, t be a term of the form $t = Y$, $t = c(Y_1, \ldots, Y_n)$ or $t = f(Y_1, \ldots, Y_n)$ and A be an abstraction. Then for all $\langle \sigma, \rho \rangle \in \gamma(A)$ and all unifiers $\langle \sigma', \rho' \rangle$ for $\sigma(X)$ and $\sigma(t)$, $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle \in \gamma(au(A, X, t))$.*

5 Conclusions and related work

In this paper we have considered an operational mechanism for the integration of functions into logic programs. This mechanism, called residuation, extends the standard unification algorithm used in SLD-resolutions by delaying unifications between unevaluable function calls and other terms. If all variables of a delayed function call are bound to ground terms, then this function call is evaluated in order to verify the delayed unification. This residuation principle yields a nice operational behaviour for many functional logic programs but has two disadvantages. One problem is that the answer to a query may contain unsolved and complex residuations for which the user cannot easily decide their solvability. A further problem is that the search space of a residuating logic program can be infinite in contrast to the equivalent logic program. This case can occur if the residuation principle generates more and more residuations which are simultaneously not solvable. Hence it is important to check at compile time whether or not this case can occur at run time. Since this is undecidable in general, we have presented an approximation to this problem based on the abstract interpretation of residuating logic programs. Our algorithm manages information about all possible residuations together with their argument variables and the dependencies between different variables in order to compute groundness information. Hence the algorithm is able to infer which residuations can be completely solved at run time.

We can also interpret our algorithm as an attempt to compile functional logic programs from languages with a complete but often complex operational semantics (e.g., EQLOG [10], SLOG [9], BABEL [16], or ALF [11]) into a more efficient execution mechanism without loosing completeness. For this purpose we check a given functional logic program by our algorithm. If the algorithm computes an abstraction containing no potential residuations, we can safely execute the program with the residuation principle. Otherwise we must apply the nondeterministic narrowing principle to compute all answers. This method can also be applied to individual parts of the program so that some parts are executed by residuation and other parts by narrowing.

The operational semantics considered in this paper originates from Le Fun [3]. The unification procedure is very similar to S-unification [4]. However, S-unification immediately reports an error if some residuations cannot be evaluated after the unification of a literal with a clause head. E.g., the example programs in section 2 and 3.3 cannot be evaluated using S-unification. Therefore Boye has extended this framework to computation with delayed residuations [6]. He has also characterized a class of operationally complete programs based on notions from attribute grammars. Compared to our abstract interpretation procedure, Boye's characterization is mainly based on the syntactic structure of the program while we have tried to approximate the operational behaviour. Hence we obtain positive results for programs where Boye's check fails. E.g., our method yields a positive answer to the completeness question of the program

```
p(A,A+A).
p(A+A,A).
```

w.r.t. the initial goal `p(2+2,1+1)` while Boye's check fails (since there are external functors in input positions).

Marriott, Søndergaard and Dart [15] have also presented an abstract interpretation algorithm for analysing logic programs with delayed evaluation. The purpose of their work was to check logic programs with negation for floundering, i.e., whether a delayed evaluation of negated subgoals is complete. This has some similarities to our framework but it is a simpler problem because a delayed evaluation of a negated literal cannot bind any goal variables since this literal is evaluated only if all arguments are ground. In our context it is important that a delayed evaluation of a residuation can bind variables in order to enable the evaluation of other residuations (see the example in Section 3.3). Therefore we have to manage the dependencies between residuations and their variables in order to analyse the data flow in this case.

Since we must restrict all abstract information to a finite domain, our algorithm cannot manage all dependencies between residuations and their variables. If a residuation depends only on variables of one clause and these variables are bound to ground terms at the end of the clause, the algorithm detects the solvability of the residuation. But if a residuation depends on local variables from different clauses, then the algorithm cannot manage it and therefore it simply infers the unsolvability of this residuation. It seems to be possible to improve the algorithm at this point by refining the abstract domain (which makes the definition of the concretisation function and the correctness proofs more complex).

Another interesting topic for further research is the question whether it is possible to adapt our proposed method to the abstract interpretation of other logic languages which are not based on SLD-resolution with the leftmost selection rule. Such a method could be applied to analyse logic programs with delay primitives.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] H. Aït-Kaci. An Overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pp. 42–58. Springer LNCS 504, 1990.

[3] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. 4th IEEE Int. Symposium on Logic Programming*, pp. 17–23, 1987.

[4] S. Bonnier. Unification in Incompletely Specified Theories: A Case Study. In *Mathematical Foundations of Computer Science*, pp. 84–92. Springer LNCS 520, 1991.

[5] P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proc. 4th IEEE Int. Symposium on Logic Programming*, pp. 318–327, 1987.

[6] J. Boye. S-SLD-resolution – An Operational Semantics for Logic Programs with External Procedures. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 383–393. Springer LNCS 528, 1991.

[7] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.

[8] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[9] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Int. Symposium on Logic Programming*, pp. 172–184, 1985.

[10] J.A. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In [8], pp. 295–363.

[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. An Abstract Interpretation Algorithm for Residuating Logic Programs. Report MPI-I-92-217, Max-Planck-Institut für Informatik, 1992.

[14] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM POPL*, pp. 111–119, Munich, 1987.

[15] K. Marriott, H. Søndergaard, and P. Dart. A Characterization of Non-Floundering Logic Programs. In *Proc. of the 1990 North American Conference on Logic Programming*, pp. 661–680. MIT Press, 1990.

[16] 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.

[17] L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 15–26. Springer LNCS 528, 1991.

[18] 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.

[19] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Research Report 12, DEC Paris Research Laboratory, 1991.

[20] P.A. Subrahmanyam and J.-H. You. FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming. In [8], pp. 157–198.