ANALYSIS OF RESIDUATING LOGIC PROGRAMS

MICHAEL HANUS

▷ 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 order to detect such situations at compile time, we present an abstract interpretation algorithm for this kind of programs. The algorithm approximates the possible residuations and instantiation states of variables during program execution. If the algorithm computes an empty residuation set for a goal, then it is ensured that the concrete execution of the goal does not end with a nonempty set of residuations which cannot be evaluated due to insufficient instantiation of argument variables. ◁

1. INTRODUCTION

Many proposals for the integration of functional and logic programming languages have been made during recent years (see [16] for a survey). 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 (ALF [12], BABEL [23], EQLOG

Address correspondence to M. Hanus, Informatik II, RWTH Aachen, D-52056 Aachen, Germany. Email: `hanus@informatik.rwth-aachen.de`

Received January 1993; accepted December 1994.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

[11], K-LEAF [6], SLOG [10], 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 [27], Le Fun [3], LIFE [2], NUE-Prolog [24], among others). The former approaches are complete under some well-defined conditions (e.g., confluence 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 process 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] and extended to constraint logic programming in [26]. For instance, consider the program (we write residuating logic programs in the usual Prolog syntax [9], 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. Therefore, 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. Hence, 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` [13]. 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:

Current goal:	*Current residuation:*
`rev([a,b,c],R)`	$\emptyset$
`a(LE1,[E1])=[a,b,c], rev(LE1,LR1)`	$\emptyset$
`rev(LE1,LR1)`	`a(LE1,[E1])=[a,b,c]`
`a(LE2,[E2])=LE1, rev(LE2,LR2)`	`a(LE1,[E1])=[a,b,c]`
`rev(LE2,LR2)`	`a(LE1,[E1])=[a,b,c], a(LE2,[E2])=LE1`
`a(LE3,[E3])=LE2, rev(LE3,LR3)`	`a(LE1,[E1])=[a,b,c], a(LE2,[E2])=LE1`
...	

Figure 1.1. Infinite derivation with the residuation principle (`a(···)` denotes `append(···)`).

```
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 (see Figure 1.1).[1] On the other hand, a functional logic language based on the narrowing principle can solve this goal and has a finite search space [13]. Therefore, we should use narrowing instead of residuation in this example.

The last example raises the important question of 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 losing any answers. A simple criterion to the completeness of residuation is the *groundness 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 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, 8, 25]) depend on SLD-resolution as the operational semantics. Hence, we cannot directly apply these frameworks to our case. But we will show that it is possible to develop a similar technique by considering unsolved residuations as part of the current substitution.

This paper is a revised and extended version of [14]. Here, we use a simplified and smaller abstract domain for the analysis. In the next section, we give a detailed

[1] A residual function call is only evaluated if all arguments are ground terms [3]. If we weaken this condition to "a residual function call is evaluated if the arguments are *sufficiently* instantiated so that exactly one defining rule is applicable" (if functions are defined by equations as in [24]), then we can also verify residuations like `append([],[E])=[a]`. In this case, the answer to the goal "`?- rev([a,b,c],R)`" can be computed by incremental verification of residuations, but there is also an infinite derivation path using the second clause infinitely many times.

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 proved in Section 4.

2. THE RESIDUATION PRINCIPLE

The residuation principle tries to avoid nondeterministic computations by delaying function calls until the arguments are sufficiently instantiated. 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 another term, 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 is delayed. If all variables in $t_1, \ldots, t_n$ will be bound to ground terms in the further computation process, the delayed function call $f(t_1, \ldots, t_n)$ will be immediately evaluated and replaced by its result in order to proceed with the unification process.

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. A *function call* is a term $f(t_1, \ldots, t_n)$ where f is a defined function. A *constructor term* is a term which does not contain function calls. A *ground term* is a term containing no variables. With this concept of terms that may contain function calls, we adopt all standard notions of logic programming [20] like clause, logic program, etc.

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.).

In order to provide a simple but precise definition of the residuation principle and to keep the analysis algorithm simple, we assume that all residuating logic programs are transformed into a flat form: in a *flat residuating logic program*, all predicate calls and clause heads have the form $p(X_1, \ldots, X_n)$ where all X_i are distinct variables (similarly to the example in [8]). 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 this flat form by introducing additional variables and equations. For instance, the residuating logic program

```
q :- p(X,Y,72), X = V-W, Y = V+W, pick(V,W).
p(A,B,A*B).
pick(9,3).
```

can be transformed into the following equivalent flat program:

```
q :- Z = 72,  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.
```

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

The computational universe of residuating logic programs contains constructor terms as well as unevaluated function calls. Therefore, we distinguish these different parts in substitutions. In the following, we assume that the *concrete domain of computation* $\mathcal{C}$ is not simply the set of all substitutions (as in logic programming), but a set of pairs of substitutions and residuations such that $\langle\sigma,\rho\rangle \in \mathcal{C}$ if

$$\begin{aligned}\sigma &= \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\} \\ \rho &= \{y_1 = r_1, \ldots, y_m = r_m\}\end{aligned}$$

where $t_1, \ldots, t_k$ are constructor terms and $r_1, \ldots, r_m$ are nonground[2] function calls, i.e., substitutions contain only constructor terms and function calls are contained in the residuation part. Since substitutions can also be represented by equations, we describe the unification algorithm for residuating logic programs in the style of Martelli and Montanari [22] by a set of transformation rules on pairs of equation systems $E; R$ where the first component E represents the substitution part and the second component R represents the residuation part. These transformation rules are shown in Figure 2.1. The standard transformation rules for unification are only applied to the first constructor-term component of the equation system. This emphasizes the fact that residuated function calls just "wait" for their evaluation. In order to enable the evaluation of a function call, instantiations of variables are propagated into the function calls (rule *Instantiate*). On the other hand, if a function call can be evaluated, its result is moved to the substitution part (rule *Evaluate*). Thus, the unification algorithm is responsible for solving equations between constructor terms and waking up residuations which are ready for evaluation. The equations between constructor terms and the residuations are generated during the evaluation of a residuating logic program (see below).

This unification procedure is not optimal in the sense that all possible failures are not detected, e.g., the nonunifiability of the equation system $x = 1, y = 2 \; ; \; x = f(z), y = f(z)$ is not detected. A more sophisticated algorithm can be found in [5]. However, our algorithm can be easily implemented using delay primitives and is used in practical implementations [3].

The unification algorithm is applied by transforming a given equation system until no more rules can be applied. The result of the unification algorithm is `fail` or a system of the form

$$x_1 = t_1, \ldots, x_k = t_k \; ; \; y_1 = r_1, \ldots, y_m = r_m$$

where each of the distinct variables x_i does not occur in t_j or r_j, and all r_j are unevaluable function calls.[3] Each $y_j = r_j$ is called a "*residual equation*" or simply

[2] We will sometimes also allow ground function calls r_i in intermediate steps. Since such calls will be evaluated during unification, they do not occur as a result of a unification process.

[3] This can be shown by a modification of the proofs presented in [22].

Clash: $\dfrac{c(t_1,\ldots,t_n) = d(t'_1,\ldots,t'_m), E \ ; \ R}{\texttt{fail}}$

if $c \neq d$ or $m \neq n$

Decompose: $\dfrac{c(t_1,\ldots,t_n) = c(t'_1,\ldots,t'_n), E \ ; \ R}{t_1 = t'_1,\ldots,t_n = t'_n, E \ ; \ R}$

Delete: $\dfrac{x = x, E \ ; \ R}{E \ ; \ R}$

Occur check: $\dfrac{x = t, E \ ; \ R}{\texttt{fail}}$

if $t \neq x$ and x occurs in t

Instantiate: $\dfrac{x = t, E \ ; \ y_1 = r_1,\ldots,y_m = r_m}{x = t, \sigma(E) \ ; \ y_1 = \sigma(r_1),\ldots,y_m = \sigma(r_m)}$

if x occurs in E or in some r_j but not in t and $\sigma = \{x \mapsto t\}$

Commute: $\dfrac{t = x, E \ ; \ R}{x = t, E \ ; \ R}$

if t is not a variable

Evaluate: $\dfrac{E \ ; \ y = f(t_1,\ldots,t_n), R}{E, y = t \ ; \ R}$

if $t_1,\ldots,t_n$ are ground and $f(t_1,\ldots,t_n)$ is evaluated to t

Figure 2.1. Unification algorithm for residuating logic programs.

"*residuation*," and we can also interpret the substitution/residuation pair $\langle\sigma,\rho\rangle$ with

$$\begin{aligned} \sigma &= \{x_1 \mapsto t_1,\ldots,x_k \mapsto t_k\} \\ \rho &= \{y_1 = r_1,\ldots,y_m = r_m\} \end{aligned}$$

as the result of the unification.

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 the unification described above. Since we assume that all programs are in flat form, all literals in goals have the form $X = Y$, $X = c(Y_1,\ldots,Y_n)$, $X = f(Y_1,\ldots,Y_n)$, or $p(X_1,\ldots,X_n)$. Thus, the proof of a literal is done by simply adding the equations to the first or second component of the current equation system from $\mathcal{C}$ (literals of the form $X = Y$ or $X = c(Y_1,\ldots,Y_n)$ are added to the substitution part, and literals of the form $X = f(Y_1,\ldots,Y_n)$ are added to the residuation part) and applying the unification algorithm. As an example, consider the following flat residuating logic program:

```
q :- Z=5, p(X,Y,Z), pick(X,Y).
```

```
p(A,B,C) :- C=A+B.

pick(D,E) :- D=2, E=3.
```

If the initial goal is q, then the following elements of the concrete domain are computed:

Current literal:	*Current substitution/residuation pair:*
q	$\langle\emptyset,\ \emptyset\rangle$
Z=5	$\langle\emptyset,\ \emptyset\rangle$
p(X,Y,Z)	$\langle\{\mathtt{Z}\mapsto 5\},\ \emptyset\rangle$
C=A+B	$\langle\{\mathtt{Z}\mapsto 5,\mathtt{A}\mapsto\mathtt{X},\mathtt{B}\mapsto\mathtt{Y},\mathtt{C}\mapsto 5\},\ \emptyset\rangle$
pick(X,Y)	$\langle\{\mathtt{Z}\mapsto 5,\mathtt{A}\mapsto\mathtt{X},\mathtt{B}\mapsto\mathtt{Y},\mathtt{C}\mapsto 5\},\ \{\mathtt{C=X+Y}\}\rangle$
D=2	$\langle\{\mathtt{Z}\mapsto 5,\mathtt{A}\mapsto\mathtt{X},\mathtt{B}\mapsto\mathtt{Y},\mathtt{C}\mapsto 5,\mathtt{D}\mapsto\mathtt{X},\mathtt{E}\mapsto\mathtt{Y}\},\ \{\mathtt{C=X+Y}\}\rangle$
E=3	$\langle\{\mathtt{Z}\mapsto 5,\mathtt{A}\mapsto 2,\mathtt{B}\mapsto\mathtt{Y},\mathtt{C}\mapsto 5,\mathtt{D}\mapsto 2,\mathtt{E}\mapsto\mathtt{Y},\mathtt{X}\mapsto 2\},\ \{\mathtt{C=2+Y}\}\rangle$
$\emptyset$	$\langle\{\mathtt{Z}\mapsto 5,\mathtt{A}\mapsto 2,\mathtt{B}\mapsto 3,\mathtt{C}\mapsto 5,\mathtt{D}\mapsto 2,\mathtt{E}\mapsto 3,\mathtt{X}\mapsto 2,\mathtt{Y}\mapsto 3\},\ \emptyset\rangle$

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

Logic programming with residuations also has some connections to the framework of constraint logic programming [18]. From a semantical point of view, residuations can be considered as *constraints* on substitutions. Therefore, the residuation framework could be viewed as a special case of the CLP framework where the domain is the set of Herbrand terms (with the defined functions as evaluable function symbols) and the constraints are equations between terms. However, this is not the case from an operational point of view because the CLP framework requires a constraint solver which checks the satisfiability of the accumulated constraints in each step. Since functions are user-defined, there need not exist a constraint solver deciding the satisfiability of the accumulated residuations, i.e., it may be the case that the current set of residuations is unsolvable,[4] e.g., the unsatisfiability of {append(L1,L2)=[1],append(L2,L1)=[2]} is not detected by the unification algorithms in [3, 5]. This would require a constraint solver for the defined list operations. But residuations can be interpreted as *passive constraints* [4] which are activated if the arguments are sufficiently instantiated. In fact, it is reasonable to integrate the residuation principle into the CLP paradigm [26], and this is done in some constraint logic languages to deal with hard constraints [19] (of course, constraint solvers which delay hard constraints are incomplete and, therefore, the same questions as discussed in this paper occur [15]).

Since the operational semantics of residuating logic programs is identical to Prolog except for the different notion of substitution and the different unification algorithm, we can apply abstract interpretation frameworks for Prolog to our case. In this paper, we will use Bruynooghe's framework [8]. This is possible since his framework does not depend on the concrete substitution or unification algorithm, but only on the left-to-right evaluation of literals, which is also the operational semantics presented in this section.

[4] This is the reason for the infinite derivation in the rev example of Section 1.

3. ABSTRACT INTERPRETATION OF RESIDUATING LOGIC PROGRAMS

In this section, we present a method to check 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 has a particular form, 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 behavior of residuating logic programs into an additional component of the concrete domain, we can use the well-known frameworks (e.g., [8, 25]) in a similar way. The heart of an abstract interpretation procedure is an abstract domain which approximates subsets of the concrete domain. 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: In order to decide whether a residuation can be evaluated at run time, we must know the variables in all potentially residuated function calls. Therefore, our abstract domain contains elements of the form "$f|_{\{X_1,\ldots,X_n\}}$" meaning: there may occur a residuated call to function f which can be evaluated if all variables $X_1, X_2, \ldots, X_n$ are ground.[5]

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. For instance, consider the goal

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

During unification of `D` and `A*A`, the first term cannot be evaluated since `A` is not ground. However, the groundness of `A` depends on the groundness of `B` and `C`. Thus, we deduce that the function call `A*A` can be evaluated if `B`

[5] The concrete name of the residuated function could be omitted in the abstract domain, but we have included it for the sake of readability.

and `C` are bound to ground terms. Hence, our abstract domain contains the element "`A` *if* {`B`,`C`}." In general, "X *if* V" means that variable X is bound to a ground term if all variables in V are bound to ground terms.

In our abstract interpretation algorithm, we analyze each clause occurring in the program. Therefore, the different abstractions computed in this algorithm contain only information about the variables of the different clauses. Hence, each abstraction A has a *domain* $dom(A)$ which is a set of variables occurring in some clause (or goal). All variables occurring in A must belong to $dom(A)$.

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):[6]

Element	Meaning
$X \text{ if } V$	X is ground if all variables in the variable set V are ground
$f\vert_V$	there is 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

The element "f" is the "worst case" in the algorithm. It will be used if the dependencies between a function call and its variables are too complex for a a finite representation.[7]

Obviously, $\mathcal{A}$ is finite if the set of variables and function symbols is finite. In our abstract domain, we use only program variables and functions occurring in the program. Therefore, $\mathcal{A}$ is finite in the case of a finite program. For convenience, we simply write "X" instead of "$X \text{ if } \emptyset$." Hence, an element "X" in an abstraction means that variable X is bound to a ground term.

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			
$A \cup \{Z,\ X \text{ if } V \cup \{Z\}\}$	$\longrightarrow$	$A \cup \{Z,\ X \text{ if } V\}$	
$A \cup \{Z,\ f\vert_{V \cup \{Z\}}\}$	$\longrightarrow$	$A \cup \{Z,\ f\vert_V\}$	
$A \cup \{f\vert_\emptyset\}$	$\longrightarrow$	A	
$A \cup \{X \text{ if } V_1,\ X \text{ if } V_2\}$	$\longrightarrow$	$A \cup \{X \text{ if } V_1\}$	if $V_1 \subseteq V_2$
$A \cup \{f\vert_{V_1},\ f\vert_{V_2}\}$	$\longrightarrow$	$A \cup \{f\vert_{V_2}\}$	if $V_1 \subseteq V_2$
$A \cup \{f\vert_V,\ f\}$	$\longrightarrow$	$A \cup \{f\}$	

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 assume that we *compute only with normalized abstractions* in the abstract interpretation algorithm.

[6] The precise meaning of the abstract elements will be formalized in Section 4.

[7] Our algorithm analyzes each clause separately. If a residuation depends on variables from different clauses, the worst case is introduced in order to ensure the termination of the analysis.

3.2. The Abstract Interpretation Algorithm

The abstract interpretation algorithm is based on several operations on the abstract domain. The most important operation is the abstract unification algorithm which approximates the concrete unification of equations occurring in clause bodies or goals. Abstract unification is a function $amgu(\alpha, t_1, t_2)$ 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 flat goal equations, the following definition is sufficient:[8]

$$
\begin{array}{lcl}
amgu(\perp, t_1, t_2) & = & \perp \\
amgu(A, X, X) & = & A \\
amgu(A, X, Y) & = & A \cup \{X \textit{ if } \{Y\},\ Y \textit{ if } \{X\}\} \quad \text{if } X \neq Y \\
amgu(A, X, c(Y_1, \ldots, Y_n)) & = & A \cup \{X \textit{ if } \{Y_1, \ldots, Y_n\},\ Y_1 \textit{ if } \{X\}, \ldots,\ Y_n \textit{ if } \{X\}\} \\
amgu(A, X, f(Y_1, \ldots, Y_n)) & = & A \cup \{X \textit{ if } \{Y_1, \ldots, Y_n\},\ f|_{\{Y_1, \ldots, Y_n\}}\}
\end{array}
$$

In the third and fourth equations of this definition, the dependencies between the variables on the left- and right-hand side are added to the current abstraction. In the last equation, only the dependency from the variables in the function call is added. The symmetric dependency would be false in general since the groundness of X in equation $X = f(Y, Z)$ does not imply the groundness of Y or Z since f may not be evaluable. In the last case, the potential residuation is also added to the current abstraction.

The next 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:

$$
\begin{array}{lcl}
call\_restrict(\perp, W) & = & \perp \\
call\_restrict(A, W) & = & \{X \textit{ if } V \in A \mid \{X\} \cup V \subseteq W\}
\end{array}
$$

Note that only dependencies between argument variables are passed. The information about residuated function calls is omitted since this information is not relevant inside the clause, but only at the end. Therefore, this information will be reconsidered at the end of the call (see below).

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

$$
\begin{array}{lcl}
exit\_restrict(\perp, W) & = & \perp \\
exit\_restrict(A, W) & = & \{X \textit{ if } V \in A \mid \{X\} \cup V \subseteq W\} \\
 & & \cup\ \{f|_V \in A \mid V \subseteq W\} \\
 & & \cup\ \{f \mid f \in A \text{ or } f|_V \in A \text{ with } V \not\subseteq W\}
\end{array}
$$

The restriction operation for clause exits transforms an abstraction element $f|_V$ 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. This is necessary to ensure the termination of the

[8]For simplicity, we omit the occur check in the abstract unification. This is safe since we compute only an approximation of the concrete unifier. Note that we always compute with normalized abstractions, i.e., the result of *amgu* will be immediately normalized.

analysis in complex cases. For the same reason, the dependency $X \textit{ if } V$ is deleted if $X \notin W$ or $V \not\subseteq W$.

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

$$\begin{aligned}
\bot \sqcup A &= A \\
A \sqcup \bot &= A \\
A_1 \sqcup A_2 &= \{X \textit{ if } V_1 \cup V_2 \mid X \textit{ if } V_1 \in A_1,\ X \textit{ if } V_2 \in A_2\} \\
&\quad \cup \{f|_V \mid f|_V \in A_1 \text{ or } f|_V \in A_2\} \\
&\quad \cup \{f \mid f \in A_1 \text{ or } f \in A_2\}
\end{aligned}$$

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 $ai(A, t = t') = amgu(A, t, t')$. 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 ($var(\xi)$ denotes the set of all variables occurring in the syntactic construction ξ):

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

Hence, a clause is interpreted in the following way. First, 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 of this call abstraction are mapped to the corresponding variables of the applied clause (A_0). Then, each literal occurring in the clause body is interpreted. The resulting abstraction (A_k) is restricted to the variables of the clause head, i.e., we forget the information about the local variables of 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 ($A_{success}$). In step 3, we compute the entire abstraction after the predicate call by combining the abstraction $A_{success}$ with the information which was forgotten by the restriction at the beginning of the predicate call (which is $A - A_{call}$).

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 [8], we construct a rational abstract AND-OR-tree representing the computation of the abstract interpretation algorithm (see also Section 4.3). 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 a 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. A detailed description of this method is given in Section 4.3.

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):

$$
\begin{array}{l}
ai(\emptyset, \texttt{q(T)}) : \\
\quad ai(\emptyset, \texttt{p(X,Y,Z)}) : \\
\qquad\qquad ai(\emptyset, \texttt{C = A*B}) = \{\texttt{C}\ \textit{if}\ \{\texttt{A,B}\},\ *|_{\{\texttt{A,B}\}}\} \\
\quad ai(\emptyset, \texttt{p(X,Y,Z)}) = \{\texttt{Z}\ \textit{if}\ \{\texttt{X,Y}\},\ *|_{\{\texttt{X,Y}\}}\} =:\ A_1 \\
\quad ai(A_1, \texttt{X = V-W}) = \{\texttt{Z}\ \textit{if}\ \{\texttt{X,Y}\},\ \texttt{X}\ \textit{if}\ \{\texttt{V,W}\},\ *|_{\{\texttt{X,Y}\}},\ -|_{\{\texttt{V,W}\}}\} =:\ A_2 \\
\quad ai(A_2, \texttt{Y = V+W}) = \{\texttt{Z}\ \textit{if}\ \{\texttt{X,Y}\},\ \texttt{X}\ \textit{if}\ \{\texttt{V,W}\},\ \texttt{Y}\ \textit{if}\ \{\texttt{V,W}\}, \\
\qquad\qquad *|_{\{\texttt{X,Y}\}},\ -|_{\{\texttt{V,W}\}},\ +|_{\{\texttt{V,W}\}}\} =:\ A_3 \\
\quad ai(A_3, \texttt{pick(V,W)}) : \\
\qquad\qquad ai(\emptyset, \texttt{A = 9}) = \{\texttt{A}\} \\
\qquad\qquad ai(\{\texttt{A}\}, \texttt{B = 3}) = \{\texttt{A, B}\} \\
\quad ai(A_3, \texttt{pick(V,W)}) = \{\texttt{V, W, Z}\ \textit{if}\ \{\texttt{X,Y}\},\ \texttt{X}\ \textit{if}\ \{\texttt{V,W}\},\ \texttt{Y}\ \textit{if}\ \{\texttt{V,W}\}, \\
\qquad\qquad *|_{\{\texttt{X,Y}\}},\ -|_{\{\texttt{V,W}\}},\ +|_{\{\texttt{V,W}\}}\} \\
\qquad\qquad \xrightarrow{normalize} \{\texttt{V, W, Z, X, Y}\} \\
ai(\emptyset, \texttt{q(T)}) = \{\texttt{T}\}
\end{array}
$$

Hence, the computed success abstraction is $\{\texttt{T}\}$. This means 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].

4. CORRECTNESS OF THE ABSTRACT INTERPRETATION ALGORITHM

In this section, we will prove the correctness of the presented abstract interpretation algorithm. First, we relate the abstract domain to the concrete domain by defining a concretization function. Then we will prove that the abstract operations defined in the previous section are correct w.r.t. the corresponding operations on the concrete domain. Finally, we obtain the correctness of our algorithm by simply applying Bruynooghe's framework [8].

4.1. Relating Abstractions to Concrete Values

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

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

However, this interpretation is not suitable because it does not cover the variable dependencies caused by residuations. For instance, if the terms `X` and `f(Y)` should be unified, the result of the unification algorithm is $\langle \emptyset; \{\mathtt{X} = \mathtt{f(Y)}\} \rangle$, i.e., the algorithm generates a residuation instead of binding `X` to `f(Y)`. On the abstract level the abstraction $\{\mathtt{X}\ \mathit{if}\ \mathtt{Y}\}$ is generated. Therefore, condition (∗) does not hold in this example.

In order to provide an appropriate relation between abstract and concrete values, we have to consider also the residuation component in condition (∗). Therefore, we extend the set $var(\sigma(V))$ by all variables which become ground if the residuations could be evaluated due to the groundness of variables in $var(\sigma(V))$. Since the evaluation of a residuation may cause the evaluation of another residuation, we consider the closure of this extension. Thus, we define $var_{\sigma,\rho}(V)$ as the smallest set satisfying the following conditions:

1. $var(\sigma(V)) \subseteq var_{\sigma,\rho}(V)$.
2. If $y = f(\bar{t}) \in \rho$ and $var(\bar{t}) \subseteq var_{\sigma,\rho}(V)$, then $var(\sigma(y)) \subseteq var_{\sigma,\rho}(V)$.

In the second condition and in the following sections, $\bar{t}$ denotes an argument sequence $t_1, \ldots, t_n$. For instance, if $\sigma = \emptyset$ and $\rho = \{\mathtt{X} = \mathtt{f(Y)}\}$ as in the previous example, then $var_{\sigma,\rho}(\{\mathtt{Y}\}) = \{\mathtt{X}, \mathtt{Y}\}$.

With this extension, we define the relation between abstract and concrete elements by the following concretization function $\gamma : \mathcal{A} \to 2^{\mathcal{C}}$:

$$
\begin{array}{lcl}
\gamma(\bot) & = & \emptyset \\
\gamma(A) & = & \{ \langle \sigma, \rho \rangle \in \mathcal{C} \mid 1.\ X\ \mathit{if}\ V \in A \Rightarrow var(\sigma(X)) \subseteq var_{\sigma,\rho}(V) \\
& & \qquad\qquad\qquad 2.\ y = f(\bar{t}) \in \rho \text{ with } y \in dom(A) \\
& & \qquad\qquad\qquad\quad \Rightarrow f \in A \ \text{ or } \ var(\bar{t}) \subseteq var(\sigma(V)) \text{ for some } f|_V \in A \ \}
\end{array}
$$

In the following, we say a substitution/residuation pair $\langle\sigma,\rho\rangle$ *satisfies the variable condition* X *if* $V \in A$ if condition 1 holds. Similarly, we say a *residuation* $y = f(\bar{t})$ *in* ρ *is covered by* A if condition 2 holds.

Condition 1 implies, for X *if* $V \in A$, that all variables of the current instantiation of X are ground if all variables of the current instantiation of V are ground terms. Condition 2 ensures that unevaluated function calls are covered by some element in A. Since an abstraction A can only contain information about variables in its domain, it cannot cover residuations bound to variables outside $dom(A)$. Since we are interested in information about the evaluation of *all* potential residuations, we will later explicitly prove (Theorem 4.4) that residuations connected to variables outside $dom(A)$ are also covered by the abstraction A at the end of the analysis.

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* $A \to A'$*, then* $\gamma(A) = \gamma(A')$.

PROOF. First, we show $\gamma(A) \subseteq \gamma(A')$. Let $\langle\sigma,\rho\rangle \in \gamma(A)$. We prove $\langle\sigma,\rho\rangle \in \gamma(A')$ by a case analysis on the applied normalization rule:

1. Let $A = A_0 \cup \{Z,\ X \textit{ if } V \cup \{Z\}\}$ and $A' = A_0 \cup \{Z,\ X \textit{ if } V\}$. Since the only difference between A and A' is the transformation of "X *if* $V \cup \{Z\}$" into "X *if* V," we have to show $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V)$. Since $\langle\sigma,\rho\rangle \in \gamma(A)$, $var(\sigma(Z)) = \emptyset$ and $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V \cup \{Z\})$. Since $\sigma(Z)$ is a ground term, $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V \cup \{Z\}) = var_{\sigma,\rho}(V)$.

2. Let $A = A_0 \cup \{Z,\ f|_{V\cup\{Z\}}\}$ and $A' = A_0 \cup \{Z,\ f|_V\}$. Since only the abstraction element $f|_{V\cup\{Z\}}$ is affected by this transformation, we have to show: if $y = f(\bar{t}) \in \rho$ with $y \in dom(A) = dom(A')$ and $var(\bar{t}) \subseteq var(\sigma(V \cup \{Z\}))$, then $var(\bar{t}) \subseteq var(\sigma(V))$. Since $\langle\sigma,\rho\rangle \in \gamma(A)$, $var(\sigma(Z)) = \emptyset$. Hence, $var(\bar{t}) \subseteq var(\sigma(V \cup \{Z\})) = var(\sigma(V))$.

3. Let $A = A' \cup \{f|_\emptyset\}$. If the abstraction element $f|_\emptyset$ was a relevant condition for $\langle\sigma,\rho\rangle \in \gamma(A)$, then $y = f(\bar{t}) \in \rho$ with $y \in dom(A)$ and $var(\bar{t}) \subseteq \emptyset$. Hence $f(\bar{t})$ is a ground function call which cannot occur in ρ.

4. Let $A = A_0 \cup \{X \textit{ if } V_1,\ X \textit{ if } V_2\}$, $A' = A_0 \cup \{X \textit{ if } V_1\}$, and $V_1 \subseteq V_2$. Obviously, $\langle\sigma,\rho\rangle \in \gamma(A')$ since the variable condition X *if* V_2 is omitted in A'.

5. Let $A = A_0 \cup \{f|_{V_1},\ f|_{V_2}\}$, $A' = A_0 \cup \{f|_{V_2}\}$, and $V_1 \subseteq V_2$. Obviously, $\langle\sigma,\rho\rangle \in \gamma(A')$ since each residuation in ρ which is covered by the omitted abstraction element $f|_{V_1}$ is also covered by $f|_{V_2}$.

6. Let $A = A_0 \cup \{f|_V,\ f\}$ and $A' = A_0 \cup \{f\}$. Obviously, $\langle\sigma,\rho\rangle \in \gamma(A')$ since each residuation in ρ which is covered by the omitted abstraction element $f|_V$ is also covered by the abstraction element f.

Next, we show $\gamma(A) \supseteq \gamma(A')$. Let $\langle\sigma,\rho\rangle \in \gamma(A')$. As before, we prove $\langle\sigma,\rho\rangle \in \gamma(A)$ by a case analysis on the applied normalization rule:

1. Let $A = A_0 \cup \{Z,\ X\ \mathit{if}\ V \cup \{Z\}\}$ and $A' = A_0 \cup \{Z,\ X\ \mathit{if}\ V\}$. Since $\langle\sigma,\rho\rangle \in \gamma(A')$, $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V) \subseteq var_{\sigma,\rho}(V \cup \{Z\})$. Hence, $\langle\sigma,\rho\rangle \in \gamma(A)$ because "$X\ \mathit{if}\ V \cup \{Z\}$" is the only altered abstraction element.
2. Let $A = A_0 \cup \{Z,\ f|_{V\cup\{Z\}}\}$ and $A' = A_0 \cup \{Z,\ f|_V\}$. This is similar to the first case.
3. Let $A = A' \cup \{f|_\emptyset\}$. This case is trivial since A contains the additional abstraction element "$f|_\emptyset$."
4. Let $A = A_0 \cup \{X\ \mathit{if}\ V_1,\ X\ \mathit{if}\ V_2\}$, $A' = A_0 \cup \{X\ \mathit{if}\ V_1\}$, and $V_1 \subseteq V_2$. We have to show $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V_2)$. But this is trivial because $\langle\sigma,\rho\rangle \in \gamma(A')$ implies $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V_1) \subseteq var_{\sigma,\rho}(V_2)$.
5. Let $A = A_0 \cup \{f|_{V_1},\ f|_{V_2}\}$, $A' = A_0 \cup \{f|_{V_2}\}$, and $V_1 \subseteq V_2$. Obviously, $\langle\sigma,\rho\rangle \in \gamma(A)$ since A contains the additional abstraction element $f|_{V_1}$.
6. Let $A = A_0 \cup \{f|_V,\ f\}$ and $A' = A_0 \cup \{f\}$. Obviously, $\langle\sigma,\rho\rangle \in \gamma(A)$ since A contains the additional abstraction element $f|_V$.

□

Due to this lemma, it makes no difference to use an abstraction A or the normalization of A if we want to prove a proposition like $\langle\sigma,\rho\rangle \in \gamma(A)$. We will take advantage of this property in the correctness proofs for the abstract operations (cf. Section 4.2).

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:

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

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

Proposition 4.1. $A_1 \sqcup A_2$ is a least upper bound of $A_1, A_2 \in \mathcal{A}$.

Proposition 4.2. If $A \sqsubseteq A'$, then $\gamma(A) \subseteq \gamma(A')$.

In order to ensure the termination of the analysis, all abstract operations used in the abstract interpretation algorithm must be monotone in their abstraction arguments. If this is the case, then recomputations in the AND-OR-graph (see Section 4.3) starting with greater elements leads to greater results w.r.t. $\sqsubseteq$. This property ensures the termination of the fixpoint computation for recursive calls. It is not difficult to show that all abstract operations defined in Section 3.2 are monotone. Therefore, we only state the monotonicity property of the abstract unification and the normalization process:

Proposition 4.3. The abstract operation amgu is monotone in its abstraction argument, i.e., $amgu(A, t_1, t_2) \sqsubseteq amgu(A', t_1, t_2)$ *provided that* $A \sqsubseteq A'$.

Proposition 4.4. The normalization process is monotone, i.e., if $A \sqsubseteq A'$ *and* B, B' *are the normalized abstractions of* A, A'*, then* $B \sqsubseteq B'$.

4.2. Correctness of Abstract Operations

Following the framework presented in [8], the correctness of the abstract interpretation algorithm can be proved by showing the correctness of each basic operation of the algorithm (like abstract unification, clause entry, and clause exit). *Correctness* means in this context that all concrete computations, i.e., the results of the concrete clause entry, clause exit, and unification (cf. Section 2), are subsumed by the abstractions computed by the corresponding abstract operations. In this section, we will prove the correctness of each of these operations.

First, we state an important property of our unification algorithm for residuating logic programs. The transformation rules in Figure 2.1 show that our unification algorithm is very similar to the classical unification algorithm for constructor terms, but with the difference that equations of the form $y = t$, where t is a ground constructor term, are added by rule *Evaluate*. This may cause additional instantiations compared to classical unification. The next proposition contains a more precise description of this behavior. In this proposition and in subsequent proofs, we *apply a substitution* τ *to a residuation* $\rho = \{y_1 = t_1, \ldots, y_m = t_m\}$ which is defined by $\tau(\rho) = \{y_1 = \tau(t_1), \ldots, y_m = \tau(t_m)\}$, i.e., the substitution is only applied to the residuated function calls. This is motivated by the special instantiation rule in Figure 2.1.

Proposition 4.5. Let t_1 *and* t_2 *be constructor terms and* $\langle\sigma, \rho\rangle \in \mathcal{C}$. *If the application of the transformation rules in Figure 2.1 to the equational representation of* $\langle\sigma, \rho\rangle$ *and the equation* $t_1 = t_2$ *yields the substitution/residuation pair* $\langle\sigma', \rho'\rangle$ *(and not* `fail`*), then*

1. $\sigma' = \tau \circ \sigma$ *with* $\sigma'(t_1) = \sigma'(t_2)$ *for some substitution* τ
2. $\rho' \subseteq \tau(\rho)$ *and* $var(\sigma'(y)) = var(\overline{t}) = \emptyset$ *for all* $y = f(\overline{t}) \in \tau(\rho) - \rho'$.

Hence, the unification algorithm for residuating logic programs computes a unifier (not necessarily a most general one) for constructor terms and may delete (i.e., evaluate) some residuations. This is the basis to prove the correctness of *amgu*, but for the complete proof, we need the following propositions about the set $var_{\sigma,\rho}(V)$.

Lemma 4.2. Let $var(\sigma(X)) \subseteq var_{\sigma,\rho}(V)$ *and* τ *be a substitution. Then* $var(\tau(\sigma(X))) \subseteq var_{\tau\circ\sigma,\tau(\rho)}(V)$.

Proof. Consider the computation of the closure $var_{\sigma,\rho}(V)$. By definition of this closure, there is a sequence $W_1, W_2, \ldots, W_n$ of variable sets with

1. $W_1 = var(\sigma(V))$,
2. $W_{i+1} = W_i \cup var(\sigma(y_i))$ for some residuation $y_i = t_i \in \rho$ with $var(t_i) \subseteq W_i$,

3. $var(\sigma(X)) \subseteq W_n$.

We define a second sequence $W'_1, W'_2, \ldots, W'_n$ of variable sets by $W'_i := var(\tau(W_i))$ $(i = 1, \ldots, n)$. This sequence has the following properties:

1. $W'_1 = var(\tau(W_1)) = var(\tau(\sigma(V)))$

2. $W'_{i+1} = W'_i \cup var(\tau(\sigma(y_i)))$ for the residuation $y_i = \tau(t_i) \in \tau(\rho)$ with $var(\tau(t_i)) \subseteq W'_i$

3. $var(\tau(\sigma(X))) \subseteq W'_n$

Hence, $var(\tau(\sigma(X))) \subseteq var_{\tau\circ\sigma,\tau(\rho)}(V)$. □

The next lemma shows that the set $var_{\sigma,\rho}(V)$ is not influenced by the evaluation of ground function calls.

Lemma 4.3. Let $\rho' \subseteq \rho$ and $var(\sigma(y)) = var(\overline{t}) = \emptyset$ for all $y = f(\overline{t}) \in \rho - \rho'$. Then $var_{\sigma,\rho}(V) = var_{\sigma,\rho'}(V)$

Proof. If some residuation element $y = f(\overline{t})$ from $\rho - \rho'$ is used to compute the closure $var_{\sigma,\rho}(V)$, it cannot add any new variable to this set since $var(\sigma(y)) = \emptyset$. Therefore, the closures $var_{\sigma,\rho}(V)$ and $var_{\sigma,\rho'}(V)$ are identical. □

Now, we can prove the correctness of $amgu$, i.e., we show that abstract unification covers all possible results of the concrete unification algorithm.

Theorem 4.1 (Correctness of Abstract Unification). Let X be a variable, t be a term of the form Y, $c(Y_1, \ldots, Y_n)$ or $f(Y_1, \ldots, Y_n)$, and A be an abstraction. Then for all $\langle\sigma, \rho\rangle \in \gamma(A)$ and all unifiers $\langle\sigma', \rho'\rangle$ computed by the rules of Figure 2.1 w.r.t. $\langle\sigma, \rho\rangle$ and $X = t$, $\langle\sigma', \rho'\rangle \in \gamma(amgu(A, X, t))$.

Proof. Let A, $\langle\sigma, \rho\rangle$, and $\langle\sigma', \rho'\rangle$ be given as described above. We prove the theorem for each of the three cases for t.
Let $t = Y$ ($\neq X$; otherwise, the theorem is trivially true). Then

$$A' := amgu(A, X, Y) = A \cup \{X \mathit{if} \{Y\},\ Y \mathit{if} \{X\}\}$$

By Proposition 4.5, $\sigma' = \tau \circ \sigma$ with $\sigma'(X) = \sigma'(Y)$ and $\rho' \subseteq \tau(\rho)$. We have to show: $\langle\sigma', \rho'\rangle \in \gamma(A')$.

1. Since $\sigma'(X) = \sigma'(Y)$, $var(\sigma'(X)) = var(\sigma'(Y))$. Therefore, $\langle\sigma', \rho'\rangle$ satisfies the variable conditions $X \mathit{if} \{Y\}$ and $Y \mathit{if} \{X\}$.

2. $Z \mathit{if} V \in A' \cap A$: Since $\langle\sigma, \rho\rangle \in \gamma(A)$, $var(\sigma(Z)) \subseteq var_{\sigma,\rho}(V)$, which implies $var(\sigma'(Z)) \subseteq var_{\sigma',\rho'}(V)$ by Proposition 4.5 and Lemmas 4.2 and 4.3.

3. $y = f(\overline{t}) \in \rho'$ with $y \in dom(A') = dom(A)$: Hence, there is a residuation $y = f(\overline{s}) \in \rho$ with $\tau(\overline{s}) = \overline{t}$. Since $\langle\sigma, \rho\rangle \in \gamma(A)$, $f \in A$ (which is the trivial case) or $f|_V \in A$ with $var(\overline{s}) \subseteq var(\sigma(V))$. The latter case implies $f|_V \in A'$ and $var(\overline{t}) = var(\tau(\overline{s})) \subseteq var(\tau(\sigma(V))) = var(\sigma'(V))$.

Next, we consider the case $t = c(Y_1, \ldots, Y_n)$. Then

$$\begin{array}{rcl} A' & := & amgu(A, X, c(Y_1, \ldots, Y_n)) \\ & = & A \cup \{X \textit{ if } \{Y_1, \ldots, Y_n\},\ Y_1 \textit{ if } \{X\}, \ldots,\ Y_n \textit{ if } \{X\}\} \end{array}$$

By Proposition 4.5, $\sigma'(X) = \sigma'(c(Y_1, \ldots, Y_n))$, which implies $var(\sigma'(X)) = var(\sigma'(c(Y_1, \ldots, Y_n)))$. Therefore, $\langle \sigma', \rho' \rangle$ satisfies the variable conditions added to A. The remaining conditions for $\langle \sigma', \rho' \rangle \in \gamma(A')$ can be proved similarly to case $t = Y$.
Now, we consider the final case $t = f(Y_1, \ldots, Y_n)$. Then

$$A' := amgu(A, X, f(Y_1, \ldots, Y_n)) = A \cup \{X \textit{ if } \{Y_1, \ldots, Y_n\},\ f|_{\{Y_1, \ldots, Y_n\}}\}$$

If $\sigma(f(Y_1, \ldots, Y_n))$ is a ground function call, it is evaluated to a ground constructor term t', and the unification algorithm simply adds the equation $X = t'$ to the first component of the equation system and the residuation component is not changed. Thus, Proposition 4.5 is applicable and the correctness of *amgu* can be shown similarly to case $t = Y$.

Now, we assume that $\sigma(f(Y_1, \ldots, Y_n))$ is not a ground function call. In this case, the unification algorithm simply adds the residuation $X = \sigma(f(Y_1, \ldots, Y_n))$, i.e., $\sigma' = \sigma$ and $\rho' = \rho \cup \{X = \sigma(f(Y_1, \ldots, Y_n))\}$. We have to show: $\langle \sigma', \rho' \rangle \in \gamma(A')$.

1. $X \textit{ if } \{Y_1, \ldots, Y_n\} \in A'$: Since $X = \sigma(f(Y_1, \ldots, Y_n)) \in \rho'$, $var(\sigma(X)) \subseteq var_{\sigma,\rho}(\{Y_1, \ldots, Y_n\})$. Hence, this variable condition is satisfied by $\langle \sigma', \rho' \rangle$.

2. $Z \textit{ if } V \in A' \cap A$: Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $var(\sigma(Z)) \subseteq var_{\sigma,\rho}(V)$, which implies $var(\sigma'(Z)) \subseteq var_{\sigma',\rho}(V) \subseteq var_{\sigma',\rho'}(V)$.

3. $y = f(\bar{t}) \in \rho'$ with $y \in dom(A') = dom(A)$: If $y = f(\bar{t}) \in \rho$, then this residuation must be covered by some element in $A \subseteq A'$. Otherwise, this residuation must be the new element $X = \sigma(f(Y_1, \ldots, Y_n))$ which is covered by the new abstraction element $f|_{\{Y_1, \ldots, Y_n\}} \in A'$.

□

Next, we prove that the abstract operations performed at the entry of a clause are correct w.r.t. the concrete semantics.

Theorem 4.2 (Correctness of Clause Entry). Let $P = p(X_1, \ldots, X_n)$ be a predicate call with abstraction A and $\langle \sigma, \rho \rangle \in \gamma(A)$. Let L :- B be a (renamed) clause, $\langle \sigma', \rho' \rangle$ be a unifier computed by the rules of Figure 2.1 w.r.t. $\langle \sigma, \rho \rangle$ and the equation $L = P$, and A_0 be the abstraction computed by algorithm ai. Then $\langle \sigma', \rho' \rangle \in \gamma(A_0)$.

Proof. Let $L = p(Z_1, \ldots, Z_n)$. First of all, note that the unifier computed for the equation $L = P$ is a trivial renaming since $Z_1, \ldots, Z_n$ are new different variables. Hence, $\rho' = \rho$ and $\sigma' = \tau \circ \sigma$ with $\tau = \{Z_1 \mapsto X_1, \ldots, Z_n \mapsto X_n\}$ (all other unifiers are renamings of this).

1. $X \textit{ if } V \in A_0$: By definition of *call_restrict* and *ai*, $\{X\} \cup V \subseteq \{Z_1, \ldots, Z_n\}$ and $\tau(X) \textit{ if } \tau(V) \in A$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $var(\sigma(\tau(X))) \subseteq var_{\sigma,\rho}(\tau(V))$, which implies $var(\sigma'(X)) \subseteq var_{\sigma',\rho}(V) = var_{\sigma',\rho'}(V)$ (note that $\sigma(\tau(Z_i)) = \sigma'(Z_i)$ for $i = 1, \ldots, n$).

2. $y = f(\bar{t}) \in \rho'$ with $y \in dom(A_0)$: This case cannot occur since $dom(A_0) = \{Z_1, \ldots, Z_n\} \cup var(B)$ which is a set of new variables. Hence, ρ' cannot contain a residuation connected to one of these variables.

□

Next, we prove the correctness of the abstract clause exit operations, i.e., we show that each substitution/residuation pair which may occur at the end of a clause applied to a predicate call is covered by the abstract interpretation algorithm.

Theorem 4.3 (Correctness of Clause Exit). Let $P = p(X_1, \ldots, X_n)$ be a predicate call with abstraction $A_{in} \neq \perp$ and $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$. Let $A = ai(A_{in}, P) = A_{success} \cup (A_{in} - A_{call})$ be the abstraction after the predicate call computed by the abstract interpretation algorithm ai. Let L :- $L_1, \ldots, L_k$ be a (renamed) clause for P, and A_k be the abstraction computed for the clause end in ai. If $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$ is an extension of the initial substitution/residuation pair $\langle \sigma_{in}, \rho_{in} \rangle$ computed by applying this clause, i.e., $\sigma_k = \sigma \circ \sigma_{in}$ with $\sigma(L) = \sigma(P)$ and $\rho_k = \rho \cup \sigma(\rho_{in})$, then $\langle \sigma_k, \rho_k \rangle \in \gamma(A)$.

PROOF. Let $L = p(Z_1, \ldots, Z_n)$ and $\tau = \{X_1 \mapsto Z_1, \ldots, X_n \mapsto Z_n\}$.

1. $X \mathit{ if } V \in A$: Hence, there are two cases:
   - $X \mathit{ if } V \in (A_{in} - A_{call})$: Since $X \mathit{ if } V \in A_{in}$ and $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$, $var(\sigma_{in}(X)) \subseteq var_{\sigma_{in}, \rho_{in}}(V)$, which implies $var(\sigma_k(X)) \subseteq var_{\sigma_k, \sigma(\rho_{in})}(V) \subseteq var_{\sigma_k, \rho_k}(V)$ (by Lemma 4.2).
   - $X \mathit{ if } V \in A_{success}$: Since $A_{exit} \sqsubseteq A_{success}$, there is a set $V' \subseteq V$ with $X \mathit{ if } V' \in A_{exit}$. By definition of A_{exit}, $\tau(X) \mathit{ if } \tau(V') \in A_k$ and $\{\tau(X)\} \cup \tau(V') \subseteq \{Z_1, \ldots, Z_n\}$. Since $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$, $var(\sigma_k(\tau(X))) \subseteq var_{\sigma_k, \rho_k}(\tau(V'))$. Since $\sigma_k(L) = \sigma_k(P)$, this implies $var(\sigma_k(X)) \subseteq var_{\sigma_k, \rho_k}(V') \subseteq var_{\sigma_k, \rho_k}(V)$.

2. $y = f(\bar{t}) \in \rho_k$ with $y \in dom(A)$: Since variables from the clause are not contained in the domain of A, the residuation $y = f(\bar{t})$ cannot be added during the processing of the clause. Hence, $y = f(\bar{t}) \in \sigma(\rho_{in})$. Thus, there is a residuation $y = f(\bar{s}) \in \rho_{in}$ with $\sigma(\bar{s}) = \bar{t}$. Since $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$ and $X \in dom(A) = dom(A_{in})$, $f \in A_{in}$ (which implies $f \in A$) or $f|_V \in A_{in}$ (which implies $f|_V \in A$) with $var(\bar{s}) \subseteq var(\sigma_{in}(V))$. In the latter case, we have $var(\bar{t}) = var(\sigma(\bar{s})) \subseteq var(\sigma(\sigma_{in}(V))) = var(\sigma_k(V))$.

□

4.3. Correctness of the Abstract Interpretation Algorithm

In the previous section, we have proved the local correctness of the basic operations of the abstract interpretation algorithm. We can combine these results into a correctness proof for the whole algorithm by using Bruynooghe's framework [8]. In his framework, the abstract interpretation algorithm generates an abstract AND-OR-tree which represents all concrete computations. To avoid infinite paths, this tree is a rational AND-OR-tree, i.e., if a predicate call is identical to (a variant of) a predicate call in an ancestor node, then this call node is identified with the

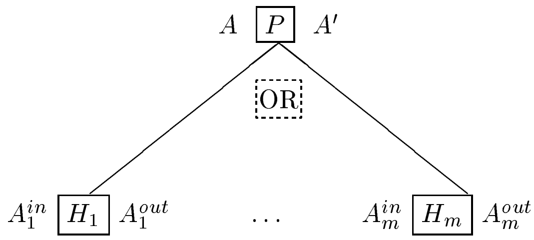

Figure 4.1. OR-node for clause entry.

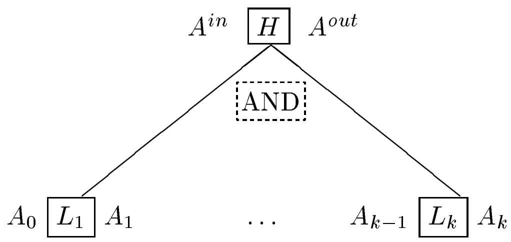

Figure 4.2. AND-node for a clause.

ancestor node. The monotonicity property of all abstract operations together with the finite domain avoids an infinite computation in this graph. Next, we will give a more detailed description of the abstract interpretation algorithm.

The abstract interpretation procedure generates the abstract AND-OR-graph as follows. In the first step, the root is created. It is marked with the initial goal (w.l.o.g. we assume that the initial goal contains only one literal) and the call abstraction for this goal. Then, this initial graph is extended by computing the success abstraction for this goal. The success abstraction A' of an equation $t = t'$ with call abstraction A is computed by abstract unification, i.e., $A' = amgu(A, t, t')$. To compute the success abstraction A' of a node with predicate call P and call abstraction A, we distinguish the following cases:

1. There is no ancestor node with the same predicate call and the same call abstraction (up to renaming of variables): First of all, we add an OR-node as shown in Figure 4.1 ($H_1, \ldots, H_m$ are the heads of all clauses for P). A_i^{in} is the call abstraction computed by our abstract operations for the entry of clause H_i :- $\cdots$ (i.e., A_0 in algorithm *ai* in Section 3.2). Then, for each new clause head H, an AND-node is added as shown in Figure 4.2 where H :- $L_1, \ldots, L_k$ is the corresponding clause. After copying the call abstraction of the head to the call abstraction of the first body literal ($A_0 = A^{in}$), the success abstraction of each literal in the clause body is computed. Then the success abstraction A^{out} of the entire clause is calculated by restricting A_k to the head variables (i.e., A^{out} is identical to A_{out} in algorithm *ai* in Section 3.2). When all success abstractions of all clauses for the predicate call P are computed, they are renamed, combined by the least upper bound operation, and then combined with the elements of A not contained in the

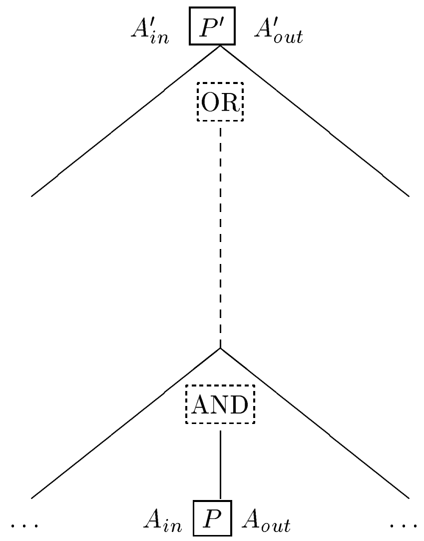

Figure 4.3. Recursive call: P is a renaming of P' and A_{in} restricted to call P is a renaming of A'_{in} restricted to call P'.

call abstraction of A (compare algorithm ai).

2. There is an ancestor node P' with the same predicate call and the same call abstraction (up to renaming of variables) (Figure 4.3): Then the success abstraction of P' (A'_{out} without the elements of A'_{in} not passed to the call P', i.e., $A_{success}$ in algorithm ai in Section 3.2) is taken as the success abstraction of P (or $\perp$ if it is not available). The combination of this success abstraction with the elements of A_{in} not contained in the call abstraction of P yields A_{out} (step 3 of algorithm ai), and we proceed with the abstract interpretation procedure (i.e., we connect P to P'). If we reach the node P' at some point during the further computation and we compute a success abstraction for P' which differs from the old success abstraction taken for P, we recompute the success abstractions beginning at P where we take the new success abstraction of P' as new success abstraction for P. The monotonicity property of the abstract operations and the finite domain ensures that this iteration terminates.

In [8], it is shown that this algorithm computes a superset of all concrete proof trees if the abstract operations for built-ins (here: unification), clause entry, and clause exit satisfy certain correctness conditions. Theorems 4.1, 4.2, and 4.3 imply exactly these correctness conditions. Hence, we can infer the correctness of our abstract interpretation algorithm since we consider the same operational semantics (left-to-right evaluation of goals), except for the different notion of substitution and unification (which does not influence Bruynooghe's general framework).

There is one remaining problem with our abstract interpretation algorithm. Initially, we wanted to characterize a class of residuating logic programs where all residuations can be evaluated at run time. However, if we analyze a program with our algorithm, the absence of elements of the form f and $f|_V$ in the success abstraction of the initial goal does not necessarily indicate that there are no unevaluated residuations at the end of the computation. Due to the definition of our concretization function γ, it may be the case that there are residuations connected to variables which are local to some clauses. The next theorem shows that this case cannot occur since *all* potential residuations are covered by our algorithm.

Theorem 4.4 (Completeness of Residuation Covering). Let L be a flat literal with abstraction A and $A' = ai(A, L)$. Let $\langle \sigma_0, \emptyset \rangle \in \gamma(A)$ and $\langle \sigma, \rho \rangle \in \gamma(A')$ be an extension of $\langle \sigma, \rho \rangle$, i.e., ρ contains the new residuations which are added during the execution of L. If $y = f(\overline{t}) \in \rho$ where $\overline{t}$ is not ground (i.e., it is a residuation which could not be evaluated), then A' contains an abstraction element of the form f or $f|_V$.

PROOF. If $y = f(\overline{t}) \in \rho$, this residuation must be generated by executing a clause containing a residuation $y = f(\overline{s})$ in the body. Since all concrete proof trees are represented by the abstract rational AND-OR-tree computed by the abstract interpretation algorithm (cf. [8]), this residuation must also be processed by our analysis algorithm which inserts the element $f|_{var(\overline{s})}$. From the definition of *amgu*, *exit_restrict*, $\sqcup$, and *ai*, it is obvious that this delay element will never be deleted in the subsequent (success) abstractions. The only possibility to delete a delay element is an application of a normalization rule, but this cannot happen if $\overline{t}$ is not ground due to the correctness of the normalization rules (Lemma 4.1). Therefore, this delay element or a transformed version of it (by operation *exit_restrict* or renaming) is contained in A'. $\Box$

Due to this theorem, our abstract interpretation algorithm characterizes a class of residuating logic programs (those containing no new elements of the form f and $f|_V$ in the success abstraction of the goal) for which all residuations can be evaluated at run time. A concrete example for the construction of an abstract AND-OR-tree will be shown in the next section.

4.4. A Final Example

The following residuating logic program is an example for a recursive procedure which requires the construction of the abstract AND-OR-tree described in the previous section. The following clauses define a predicate `sum(L,S)` which computes the sum `S` of a list of numbers `L`:

```
sum([],0).

sum([E|R],E+RS) :- sum(L,RS).
```

For instance, the execution of the goal `sum([1,3,5],S)` yields the answer `S=9`. The concrete computation is shown in the following table:

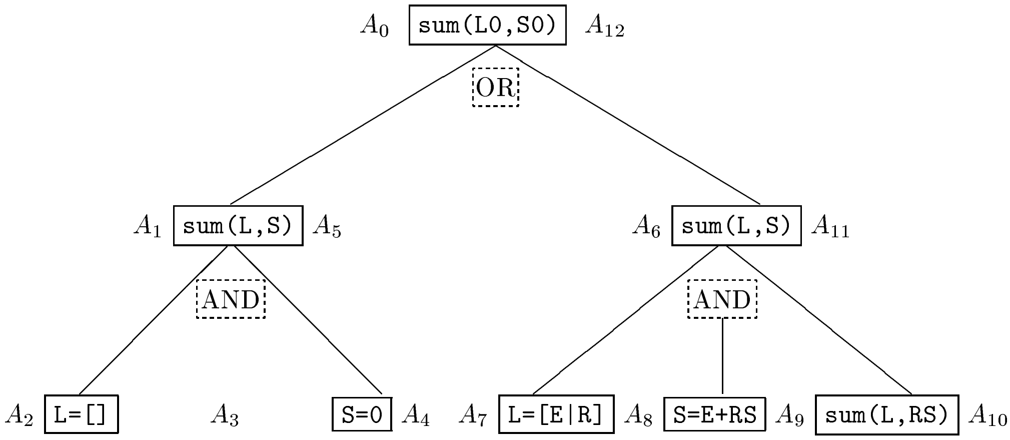

Figure 4.4. AND-OR-tree for the abstract interpretation of `sum(L0,S0)`.

Goal	Current Residuation	Current Substitution
`sum([1,3,5],S)`	$\emptyset$	$\emptyset$
`sum([3,5],RS1)`	{`S=1+RS1`}	$\emptyset$
`sum([5],RS2)`	{`S=1+RS1, RS1=3+RS2`}	$\emptyset$
`sum([],RS3)`	{`S=1+RS1, RS1=3+RS2, RS2=5+RS3`}	$\emptyset$
$\emptyset$	$\emptyset$	{`RS3`$\mapsto$`0, RS2`$\mapsto$`5, RS1`$\mapsto$`8, S`$\mapsto$`9`}

We want to show that the residuation principle computes a fully evaluated answer for `S` for any given list of numbers `L`. In order to apply our abstract interpretation algorithm, we transform the program into an equivalent flat program:

```
sum(L,S) :- L=[], S=0.
sum(L,S) :- L=[E|R], S=E+RS, sum(L,RS).
```

The initial goal is `sum(L0,S0)` with abstraction {`L0`}, i.e., it is a predicate call with a ground first argument. Our abstract interpretation algorithm applied to this goal and abstraction generates the abstract AND-OR-tree shown in Figure 4.4. We will see that the tree is finite because the literal `sum(L,RS)` together with the call abstraction part of A_9 is a renaming of the root literal `sum(L0,S0)` together with the call abstraction part of A_0. In the following, we describe the computation of the abstract interpretation algorithm and the evolving values of the abstractions A_i.

- $A_0 = \{$`L0`$\}$: The call abstraction of the root literal is the initial abstraction of the goal.
- $A_1 = \{$`L`$\}$ and $A_6 = \{$`L`$\}$: The root is an OR-node with two sons since two clauses can be applied to the literal `sum(L0,S0)`. The entry abstractions for these clauses are computed from A_0 by *call_restrict* and renaming.
- $A_2 = \{$`L`$\}$: The entry abstraction of the clause is also the abstraction for the first predicate call in the clause body.

- $A_3 = \{\mathtt{L}\}$: The abstraction A_2 is not modified by abstract unification since `L` is already ground.
- $A_4 = \{\mathtt{L},\ \mathtt{S}\}$: `S` is added to the abstraction by abstract unification since it is bound to a ground term after this unification.
- $A_5 = \{\mathtt{L},\ \mathtt{S}\}$: The exit abstraction of this clause is the exit abstraction of the last body literal restricted to the variables in the clause head.
- $A_7 = \{\mathtt{L}\}$: The entry abstraction of the second clause is also the abstraction for the first predicate call in the clause body.
- $A_8 = \{\mathtt{L},\ \mathtt{E},\ \mathtt{R}\}$: The variables `E` and `R` are ground since `L` is ground. This is computed by the abstract unification algorithm together with the normalization rules.
- $A_9 = \{\mathtt{L},\ \mathtt{E},\ \mathtt{R},\ \mathtt{S}\,\mathit{if}\,\{\mathtt{RS}\},\ \mathtt{+}|_{\{\mathtt{RS}\}}\}$: The function call to `+` is added to the abstraction. It cannot be evaluated until the variable `RS` is ground.
- $A_{10} = \perp$: The call abstraction part of A_9 is $\{\mathtt{L}\}$ (compare definition of *call_restrict*). Hence, this predicate call is a renaming of the predicate call at the root. Therefore, we take the value $\perp$ as the success abstraction for this call since the success abstraction of the root call is not yet known. However, if the latter success abstraction is available and different from $\perp$, we start a recomputation at this point.
- $A_{11} = \perp$: The exit abstraction of the second clause is the exit abstraction of the last body literal.
- $A_{12} = \{\mathtt{L0},\ \mathtt{S0}\}$: The success abstraction of the root predicate call is the least upper bound of $\{\mathtt{L0},\ \mathtt{S0}\}$ and $\perp$ together with the elements of A_0 not contained in the call abstraction (actually, there are no such elements). Since the success abstraction of the root call is now available and different from $\perp$, we restart the evaluation of the abstraction A_{10}.
- $A_{10} = \{\mathtt{L},\ \mathtt{RS},\ \mathtt{E},\ \mathtt{R},\ \mathtt{S}\}$: The new value of A_{10} is computed from the new renamed success abstraction of the root predicate call ($\{\mathtt{L},\mathtt{RS}\}$) together with the elements of A_9 not contained in the call abstraction giving $\{\mathtt{L},\ \mathtt{RS},\ \mathtt{E},\ \mathtt{R},\ \mathtt{S}\,\mathit{if}\,\{\mathtt{RS}\},\ \mathtt{+}|_{\{\mathtt{RS}\}}\}$. This abstraction, simplified by the normalization rules, is the new value of A_{10}.
- $A_{11} = \{\mathtt{L},\ \mathtt{S}\}$: The exit abstraction of the second clause is the exit abstraction of the last body literal restricted to the variables in the clause head.
- $A_{12} = \{\mathtt{L0},\ \mathtt{S0}\}$: The success abstraction of the root predicate call is the least upper bound of the renamed exit abstractions A_5 and A_{11} (which are identical). Since the success abstraction of the root call is identical to the previous value, we need not restart the evaluation of the abstraction A_{10}. Hence, the abstract interpretation algorithm is finished.

Since the abstract interpretation algorithm has computed the exit abstraction $\{\mathtt{L0},\ \mathtt{S0}\}$ for the initial goal, we conclude by the correctness of the abstract interpretation algorithm and Theorem 4.4 that variable `S0` is bound to a ground term, and there are no unevaluable residuations at the end of a successful computation.

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-resolution 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 behavior 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., ALF [12], BABEL [23], EQLOG [11], or SLOG [10]) 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, then we can safely execute the program with the residuation principle, i.e., all valid answers are computed by the residuation principle (provided that the computation terminates). 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 using the residuation principle and other parts are executed by narrowing. For instance, in order to avoid the termination problem in the "reverse" example in Section 1, we can check the solvability of the residuated function calls by narrowing just before the recursive call to `rev`. Our algorithm can be simply modified to compute the necessary information to decide at compile time whether there may be residuated functions before recursive predicate calls at run time.

The operational semantics considered in this paper originates from Le Fun [3]. The unification procedure is very similar to S-unification [5]. 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 [7]. 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 behavior. 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 et al. [21] have also presented an abstract interpretation algorithm for analyzing 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 problem has similarities to our residuation problem, but it is also very different due to the following reasons:

1. A delayed evaluation of a negated literal cannot bind any goal variables since this literal is evaluated 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 analyze the data flow in this case.

2. In our context, the terms contain constructors *and* function calls. The right abstraction of these terms complicates the correctness proofs of our algorithm.

On the other hand, we cannot analyze logic programs with delayed negation with our algorithm (for instance, by declaring all negated literals as functions) since we consider the evaluation of a ground function call as an atomic operation. However, the evaluation of a negated literal may cause the evaluation of other negated literals, i.e., it is not an atomic operation. Nevertheless, it would be interesting to extend our algorithm to a more detailed analysis of function calls if the functions are specified and evaluated in a particular formalism (for instance, by conditional equations as in ALF [12]).

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. However, if a residuation depends on local variables from different clauses, then the algorithm cannot manage it, and simply infers the unsolvability of this residuation. It would be interesting to improve the algorithm at this point by refining the abstract domain.

Another interesting topic for further research is the question of 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 analyze the floundering problem of NU-Prolog or to derive run-time properties of the Andorra computation rule [17].

The author is grateful to the anonymous referees for their suggestions to improve the analysis and the readability of this paper. The research described in this paper was made during the author's stay at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. It was supported in part by the German Ministry for Research and Technology

(BMFT) under Grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics). The responsibility for the contents of this publication lies with the author.

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 Internat. Symposium on Logic Programming*, pp. 17–23, San Francisco, 1987.

4. H. Aït-Kaci and A. Podelski. Functions as Passive Constraints in LIFE. Research Report 13, DEC Paris Research Laboratory, 1991.

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

6. 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 Internat. Symposium on Logic Programming*, pp. 318–327, San Francisco, 1987.

7. 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.

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

9. W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, third rev. and ext. edition, 1987.

10. 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.

11. J.A. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 295–363. Prentice Hall, 1986.

12. 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.

13. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.

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. Analysis of Nonlinear Constraints in CLP($\mathcal{R}$). In *Proc. Tenth International Conference on Logic Programming*, pp. 83–99. MIT Press, 1993.

16. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

17. S. Haridi and P. Brand. Andorra Prolog: An Integration of Prolog and Committed Choice Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 745–754, 1988.

18. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.

19. J. Jaffar, S. Michaylov, and R.H.C. Yap. A Methodology for Managing Hard Constraints in CLP Systems. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 306–316. SIGPLAN Notices, Vol. 26, No. 6, 1991.

20. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

21. 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.

22. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

23. 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.

24. 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.

25. 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.

26. G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pp. 405–419. MIT Press, 1993.

27. P.A. Subrahmanyam and J.-H. You. FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 157–198. Prentice Hall, 1986.