# MAX-PLANCK-INSTITUT FÜR INFORMATIK

An Abstract Interpretation Algorithm
for Residuating Logic Programs

Michael Hanus

MPI–I–92–217 April 1992


Im Stadtwald
W 6600 Saarbrücken
Germany

## Author's Address

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 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.

## Keywords

Logic Programming, Functional Logic Programming, Residuation, Abstract Interpretation

# 1 Introduction

Many proposals for the integration of functional and logic programming languages have been made during recent years (see [DL86] 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 [GM86], SLOG [Fri85], K-LEAF [BGL+87], BABEL [MR92], ALF [Han90], 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 [SY86], Le Fun [AKLN87], LIFE [AK90], NUE-Prolog [Nai91], 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 [AKLN87] and extended to constraint logic programming in [Smo91]. For instance, consider the following program (we write residuating logic programs in the usual Prolog syntax [CM87] 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

| *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: Infinite derivation with the residuation principle (`a(···)` denotes `append(···)`)

equation whereas narrowing computes the unique value for `E` [Han91]. 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 (see Figure 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 [Han91]. 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 *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 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 an abstract interpretation technique to this kind of programs. Previous approaches for abstract interpretation of logic programs (see, for instance, [AH87, Bru91, Nil90]) depends 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.

[1] A residual function call is only evaluated if all arguments are ground terms [AKLN87]. If we weaken this condition to "a residual function call is evaluated if the arguments are *sufficiently* instantiated" (as in [Nai91]), 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.

In the next section we give a detailed description of the operational semantics considered in this paper. The abstract domain and the abstract interpretation algorithm for reasoning about residuating programs is 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 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.

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 *constructor term* is a term which does not contain functions, and a *non-constructor term* is a term containing at least one function call. 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 [Llo87] like clause, logic program, substitution 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.).

The unification algorithm for residuating logic programs is described in Figure 2 by a set of transformation rules on term equations $E$ in the style of Martelli and Montanari [MM82].[2] In order to unify two terms $t$ and $t'$, we transform the equation $t = t'$ until no more rules are applicable. In this case we yield the result `fail` or an equation set of the form

$$x_1 = t_1, \ldots, x_k = t_k,\ s_1 = s'_1, \ldots, s_m = s'_m$$

[2]There is one peculiarity in the unification algorithm in the presence of defined functions. If a variable $X$ occurs in a term $t$ inside a function call, then we cannot deduce the failure of the unification problem $X = t$ due to the occur check. For instance, if $id$ is the identity function, then the equation $X = id(X)$ is valid for any value of $X$ and hence we cannot deduce a failure. But in this case $X$ cannot be further instantiated to a ground term and therefore the residuation $X = id(X)$ will never be verified or disproved. Hence a failure is generated also in this case.

| | | |
|---|---|---|
| *Clash*: | $\dfrac{c(t_1,\ldots,t_n) = d(t'_1,\ldots,t'_m), E}{\texttt{fail}}$ | if $c \neq d$ or $m \neq n$ |
| *Decompose*: | $\dfrac{c(t_1,\ldots,t_n) = c(t'_1,\ldots,t'_n), E}{t_1 = t'_1,\ldots,t_n = t'_n, E}$ | |
| *Delete*: | $\dfrac{X = X, E}{E}$ | |
| *Occur check*: | $\dfrac{X = t, E}{\texttt{fail}}$ | if $t \neq X$ and $X$ occurs in $t$ |
| *Instantiate*: | $\dfrac{X = t, E}{X = t, \sigma(E)}$ | if $X$ occurs in $E$ but not in $t$ and $\sigma = \{X \mapsto t\}$ |
| *Commute*: | $\dfrac{t = X, E}{X = t, E}$ | if $t$ is not a variable |
| *Evaluate-l*: | $\dfrac{f(t_1,\ldots,t_n) = t, E}{t' = t, E}$ | if $t_1,\ldots,t_n$ are ground and $f(t_1,\ldots,t_n)$ is evaluated to $t'$ |
| *Evaluate-r*: | $\dfrac{t = f(t_1,\ldots,t_n), E}{t = t', E}$ | if $t_1,\ldots,t_n$ are ground and $f(t_1,\ldots,t_n)$ is evaluated to $t'$ |

Figure 2: Unification algorithm for residuating logic programs

where each variable $x_i$ does not occur in $t_j$, $s_j$ or $s'_j$, and $s_i$ or $s'_i$ are unevaluable function calls ($i = 1,\ldots,m$). In the latter case $s_i = s'_i$ is called "*residual equation*" or simply "*residuation*" and we interpret the substitution/residuation pair $\langle\sigma,\rho\rangle$ with

$$\begin{aligned}\sigma &= \{x_1 \mapsto t_1,\ldots,x_k \mapsto t_k\}\\ \rho &= \{s_1 = s'_1,\ldots,s_m = s'_m\}\end{aligned}$$

as the result of the unification. In the following we consider the elements $s_i = s'_i$ of the residuation $\rho$ as multisets $\{s_i, s'_i\}$ in order to abstract from the "left" or "right" side of a residual equation (which is irrelevant in this context), i.e., we identify the residual equations $s_i = s'_i$ and $s'_i = s_i$. In the entire computation $\sigma$ is part of the answer substitution and $\rho$ will be added to unifications in subsequent resolution steps. By giving priority to the *evaluate* rules we obtain the operational semantics of Le Fun [AKLN87] where residuations are immediately verified if all argument terms are ground.

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. Hence 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 (multiset) $r = r'$ 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 $\rho$.

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

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

# 3 Abstract interpretation of residuating logic 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 is discussed in Section 4 in more detail. We assume familiarity with basic ideas of abstract interpretation techniques [AH87].

## 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 [AH87]). Since we have abstracted the different operational behaviour of residuating logic programs into an additional component to the concrete domain, we can use the well-known frameworks (e.g., [Bru91, Nil90]) 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 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 "`X with +`$|_{\{\texttt{A,B}\}}$" 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. For instance, 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 "`C if {A,B}`". 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.

**Sharing between variables:** The potential residuations can be copied between different variables in the unification process. For instance, 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 {`X,Z`} representing the sharing between `X` and `Z`.

Summarizing the previous discussion, our *abstract domain* $\mathcal{A}$ contains the element $\bot$ (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: |
|---|---|
| `X if` $V$ | `X` is ground if all variables in the variable set $V$ are ground |
| `X with f`$\|_V$ | `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 |
| {`X,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 "`X 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 "$X$ `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:** | | | |
|---|---|---|---|
| $A \cup \{Z,\ X \texttt{ if } V \cup \{Z\}\}$ | $\longrightarrow$ | $A \cup \{Z,\ X \texttt{ if } V\}$ | if $Z$ is function-free in $A$ |
| $A \cup \{Z,\ X \texttt{ with } f\|_{V \cup \{Z\}}\}$ | $\longrightarrow$ | $A \cup \{Z,\ X \texttt{ with } f\|_V\}$ | if $Z$ is function-free in $A$ |
| $A \cup \{X \texttt{ with } f\|_\emptyset\}$ | $\longrightarrow$ | $A$ | |
| $A \cup \{X \texttt{ if } V_1,\ X \texttt{ if } V_2\}$ | $\longrightarrow$ | $A \cup \{X \texttt{ if } V_1\}$ | if $V_1 \subseteq V_2$ |
| $A \cup \{X,\ \{X, Y\}\}$ | $\longrightarrow$ | $A \cup \{X\}$ | |

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 [Bru91]). 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

```
q(T) :- 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(T) :- 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 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:

$$\mathit{call\_restrict}(\bot, W) \quad = \quad \bot$$

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

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 $V = \emptyset$.[3]

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

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

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 $\mathit{call\_restrict}(A, W)$ in order to combine it after a predicate call:

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

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

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

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 $\mathit{amgu}(\alpha, t_1, t_2)$

[3] I conjecture that it is also possible to pass abstraction elements $X \,\texttt{if}\, V$ to the clause if $\{X\} \cup V \subseteq W$, but I could not prove the correctness of the abstract interpretation algorithm for this extension.

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:[4]

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

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 $\alpha$ 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:

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 $A_{call} = call\_restrict(A, \{X_1, \ldots, X_n\})$
   $A_0 = \langle$replace all $X_i$ by $Z_i$ in $A_{call}\rangle$
   $A_1 = ai(A_0, L_1)$
   $A_2 = ai(A_1, L_2)$
   $\vdots$
   $A_k = ai(A_{k-1}, L_k)$
   $A_{out} = exit\_restrict(A_k, \{Z_1, \ldots, Z_n\})$
   $A_{exit} = \langle$replace all $Z_i$ by $X_i$ in $A_{out}\rangle$
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 $A_{success} \neq \bot$, else $\bot$

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

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

$p$ 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 [Bru91] we can 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 $\bot$ 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. 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 [AKLN87]:

```
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):

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

$ai(\emptyset, \texttt{p(X,Y,Z)})$:

$ai(\emptyset, \texttt{C = A*B}) = \{\texttt{C if \{A,B\}}, \texttt{C with *}|_{\{\texttt{A,B}\}}\}$

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

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

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

$ai(A_3, \texttt{pick(V,W)})$:

$ai(\emptyset, \texttt{A = 9}) = \{\texttt{A}\}$

$ai(\{\texttt{A}\}, \texttt{B = 3}) = \{\texttt{A, B}\}$

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

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

Hence the computed success abstraction is {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 [AKLN87].

# 4 Correctness of the abstract interpretation algorithm

In this section we will prove the correctness of the presented abstract interpretation algorithm. Firstly, we relate the abstract domain to the concrete domain by defining a concretisation 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 [Bru91].

## 4.1 Relating abstractions to concrete values

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 $\gamma$ is the correct interpretation of an abstraction "X if $V$". The intuitive meaning is "the interpretation of X is ground if all interpretations of $V$ are ground". To be more precise, "X 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 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 $\xi$) Such a definition seems to justify the generation of the abstractions "X if {Y}" and "Y if {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* $\rho$.

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 $t$ (in subsequent proofs we say "*l-variable*" for variables belonging to this set):

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

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

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

(where $\bar{t}$ denotes the argument sequence $t_1, \ldots, t_n$). Note that $var_\rho(\emptyset) = \emptyset$ if $\rho$ 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 $\rho = \{\texttt{f(X)=c(Y)}, \texttt{f(X)=c(Z)}\}$, then $var_\rho(\{\texttt{X}\}) = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$. We extend the function $var_\rho$ to terms by

$$var_\rho(t) = var_\rho(var(t))$$

and 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 $\bar{t}$ denotes the argument sequence $t_1, \ldots, t_n$):

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

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

Condition 1 implies for $X\ \texttt{if}\ V \in A$ that all l-variables of 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 $\sigma$ 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 $A \to A'$, 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 $A = A_0 \cup \{Z,\ X \,\texttt{if}\, V \cup \{Z\}\}$, $A' = A_0 \cup \{Z,\ X \,\texttt{if}\, V\}$ and $Z$ be function-free in $A_0$. Since the only difference between $A$ and $A'$ is the transformation of "$X \,\texttt{if}\, V \cup \{Z\}$" into "$X \,\texttt{if}\, V$", we have to show $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V))$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $lvar(\sigma(Z)) = \emptyset$ and $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V \cup \{Z\}))$. $\sigma(Z)$ is a ground term because $Z$ is function-free in $A_0$. Hence $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V \cup \{Z\})) = var_\rho(\sigma(V))$.

2. Let $A = A_0 \cup \{Z,\ X \,\texttt{with}\, f|_{V \cup \{Z\}}\}$, $A' = A_0 \cup \{Z,\ \ X \,\texttt{with}\, f|_V\}$ and $Z$ be function-free in $A_0$. Since only the abstraction element $X \,\texttt{with}\, f|_{V \cup \{Z\}}$ is affected by this transformation, we have to show: if $f(\bar{t})$ occurs directly in $\sigma(X)$ or $\rho$ with $var(\bar{t}) \neq \emptyset$ and $var(\bar{t}) \subseteq var(\sigma(V \cup \{Z\}))$, then $var(\bar{t}) \subseteq var(\sigma(V))$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $var(\sigma(Z)) = lvar(\sigma(Z)) = \emptyset$ (as in the previous case). Hence $var(\bar{t}) \subseteq var(\sigma(V \cup \{Z\})) = var(\sigma(V))$.

3. Let $A = A' \cup \{X \,\texttt{with}\, f|_\emptyset\}$. If the abstraction element $X \,\texttt{with}\, f|_\emptyset$ was a relevant condition for $\langle \sigma, \rho \rangle \in \gamma(A)$, then $f(\bar{t})$ occurs directly in $\sigma(X)$ or $\rho$ with $var(\bar{t}) \subseteq \emptyset$. Hence $f(\bar{t})$ is a ground function call which need not be covered by $A'$.

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

5. Let $A = A_0 \cup \{X,\ \{X, Y\}\}$ and $A' = A_0 \cup \{X\}$. If the abstraction element $\{X, Y\}$ was a relevant condition for $\langle \sigma, \rho \rangle \in \gamma(A)$, then $lvar(\sigma(X)) \cap lvar(\sigma(Y)) \neq \emptyset$. But this case cannot occur since $lvar(\sigma(X)) = \emptyset$.

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 $A = A_0 \cup \{Z,\ X \,\texttt{if}\, V \cup \{Z\}\}$ and $A' = A_0 \cup \{Z,\ X \,\texttt{if}\, V\}$. Since $\langle \sigma, \rho \rangle \in \gamma(A')$, $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V)) \subseteq var_\rho(\sigma(V \cup \{Z\}))$. Hence $\langle \sigma, \rho \rangle \in \gamma(A)$ because "$X \,\texttt{if}\, V \cup \{Z\}$" is the only altered abstraction element.

2. Let $A = A_0 \cup \{Z,\ X \,\texttt{with}\, f|_{V \cup \{Z\}}\}$ and $A' = A_0 \cup \{Z,\ \ X \,\texttt{with}\, f|_V\}$. Similarly to the first case.

3. Let $A = A' \cup \{X \,\texttt{with}\, f|_\emptyset\}$. This case is trivial since $A$ contains the additional abstraction element "$X \,\texttt{with}\, f|_\emptyset$".

4. Let $A = A_0 \cup \{X \,\texttt{if}\, V_1,\ X \,\texttt{if}\, V_2\}$, $A' = A_0 \cup \{X \,\texttt{if}\, V_1\}$ and $V_1 \subseteq V_2$. We have to show $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V_2))$. But this is trival because $\langle \sigma, \rho \rangle \in \gamma(A')$ implies $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V_1)) \subseteq var_\rho(\sigma(V_2))$.

5. Let $A = A_0 \cup \{X,\ \{X, Y\}\}$ and $A' = A_0 \cup \{X\}$. This case is trivial since $A$ contains the additional abstraction element "$\{X, Y\}$".

■

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:

(a) $\bot \sqsubseteq \alpha$ for all $\alpha \in \mathcal{A}$

(b) $A \sqsubseteq A' \iff$
1. $X \,\texttt{if}\, V' \in A' \ \Rightarrow\ \exists V \subseteq V'$ 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. $\{X, Y\} \in A \ \Rightarrow\ \{X, Y\} \in A'$

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

**Lemma 4.2** *$A_1 \sqcup A_2$ is a least upper bound of $A_1, A_2 \in \mathcal{A}$.*

*Proof:* If $A_1 = \bot$ or $A_2 = \bot$, then obviously $A_1 \sqcup A_2$ is a least upper bound of $A_1$ and $A_2$. Therefore we assume $A_1 \neq \bot$ and $A_2 \neq \bot$.

First we show that $A_1 \sqcup A_2$ is an upper bound of $A_1$ (the case for $A_2$ is symmetric): Let $X \,\texttt{if}\, V_0 \in A_1 \sqcup A_2$. By definition of $\sqcup$, there are $X \,\texttt{if}\, V_1 \in A_1$ and $X \,\texttt{if}\, V_2 \in A_2$ with $V_0 = V_1 \cup V_2$. Hence $X \,\texttt{if}\, V_1 \in A_1$ and $V_1 \subseteq V_0$ (condition 1 of $\sqsubseteq$). If $\alpha$ is an abstraction element of the form $X \,\texttt{with}\, f|_V$, $f$ or $\{X, Y\}$, then $\alpha \in A_1$ implies $\alpha \in A_1 \sqcup A_2$ by definition of $\sqcup$ (conditions 2-4 of $\sqsubseteq$). Therefore $A_1 \sqsubseteq A_1 \sqcup A_2$.

To show that $A_1 \sqcup A_2$ is a least upper bound, assume an abstraction $A$ with $A_1 \sqsubseteq A$ and $A_2 \sqsubseteq A$. If $X \,\texttt{if}\, V \in A$, then there are $V_1 \subseteq V$ and $V_2 \subseteq V$ with $X \,\texttt{if}\, V_1 \in A_1$ and $X \,\texttt{if}\, V_2 \in A_2$ (by definition of $\sqsubseteq$). This implies $X \,\texttt{if}\, V_1 \cup V_2 \in A_1 \sqcup A_2$ and $V_1 \cup V_2 \subseteq V$. If $\alpha$ is an abstraction element of the form $X \,\texttt{with}\, f|_V$, $f$ or $\{X, Y\}$, then $\alpha \in A_1 \sqcup A_2$ implies $\alpha \in A_1$ or $\alpha \in A_2$ and hence $\alpha \in A$ by definition of $\sqsubseteq$. Therefore $A_1 \sqcup A_2 \sqsubseteq A$. ■

It is also easy to show that $\gamma$ is a monotone function:

**Lemma 4.3** *If $A \sqsubseteq A'$, then $\gamma(A) \subseteq \gamma(A')$.*

*Proof:* Let $A \sqsubseteq A'$ and $\langle \sigma, \rho \rangle \in \gamma(A)$. (the case $A = \bot$ is trivial). We have to show $\langle \sigma, \rho \rangle \in \gamma(A')$.

Let $X \,\texttt{if}\, V' \in A'$. Since $A \sqsubseteq A'$, there is a set $V \subseteq V'$ with $X \,\texttt{if}\, V \in A$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $lvar(\sigma(X)) \subseteq var_\rho(\sigma(V)) \subseteq var_\rho(\sigma(V'))$. Hence $\langle \sigma, \rho \rangle$ satisfies $X \,\texttt{if}\, V'$.

If $f(\bar{t})$ occurs directly in $\sigma(X)$ or $\rho$ with $var(\bar{t}) \neq \emptyset$, then $f \in A$ or $var(\bar{t}) \subseteq var(\sigma(V))$ for some $X \,\texttt{with}\, f|_V \in A$. These abstraction elements are also contained in $A'$ in both cases (by definition of $\sqsubseteq$).

If $lvar(\sigma(X)) \cap lvar(\sigma(Y)) \neq \emptyset$ for variables $X \neq Y$, then $\{X, Y\} \in A$. This implies $\{X, Y\} \in A'$ by definition of $\sqsubseteq$. ■

It is also not difficult to show that all abstract operations defined in Section 3.2 (restriction, remainder, abstract unification etc.) are monotone. As an example we show the monotonicity of the restriction operation for clause entries.

**Lemma 4.4** *The abstract operation call_restrict is monotone.*

*Proof:* Let $A_1 \sqsubseteq A_2$ and $A_1' := call\_restrict(A_1, W)$, $A_2' := call\_restrict(A_2, W)$. We have to show: $A_1' \sqsubseteq A_2'$.

If $A_1 = \bot$, then $A_1' = \bot$ and thus $A_1' \sqsubseteq A_2'$. Hence we assume $A_1 \neq \bot$ which implies $A_2 \neq \bot$ and $A_1' \neq \bot$, $A_2' \neq \bot$.

1. $X \,\texttt{if}\, V \in A_2'$: By definition of *call_restrict*, $V = \emptyset$, $X \in A_2$ and $X \in W$. Since $A_1 \sqsubseteq A_2$, $X \in A_1$ which immediately implies $X \in A_1'$.

2. $X \,\texttt{with}\, f|_V \in A_1'$: By definition of *call_restrict*, $X \,\texttt{with}\, f|_V \in A_1$ and $\{X\} \cup V \subseteq W$. This implies $X \,\texttt{with}\, f|_V \in A_2$ and thus $X \,\texttt{with}\, f|_V \in A_2'$.

3. $f \in A_1'$: By definition of *call_restrict*, either $f \in A_1$ which implies $f \in A_2$ and $f \in A_2'$, or $X \,\texttt{with}\, f|_V \in A_1$ with $X \in W$ and $V \not\subseteq W$. The latter case implies $X \,\texttt{with}\, f|_V \in A_2$ and $f \in A_2'$.

4. $\{X, Y\} \in A_1'$: By definition of *call_restrict*, $\{X, Y\} \in A_1$ and $X, Y \in W$. Hence $\{X, Y\} \in A_2$ and therefore $\{X, Y\} \in A_2'$.

■

## 4.2 Correctness of abstract operations

Following the framework presented in [Bru91], 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. In the following we use standard notions and notations from term rewriting [DJ90]. For instance, a position $\pi$ in a term $t$ is a sequence of natural numbers denoting the path from the root symbol to this term position, and $t|_\pi$ denotes the subterm of $t$ at position $\pi$.

First we prove that the abstract unification operation covers all possible concrete unifiers. For this purpose we need several propositions. A unifier of two terms containing function calls does not make these terms identical (due to the residuations), but the following lemma states a relationship between the variables of the unified terms.

**Lemma 4.5** *If $t_1$ and $t_2$ are terms and $\langle \sigma, \rho \rangle$ is a unifier computed by the rules of Figure 2, then $lvar(\sigma(t_1)) \subseteq var_\rho(\sigma(t_2))$.*

*Proof:* We prove the lemma by analysing different subterms of $\sigma(t_1)$ and $\sigma(t_2)$ which contain variables of $lvar(\sigma(t_1))$. Let $\sigma(t_1)|_\pi$ be a subterm of $\sigma(t_1)$ which is not inside a function call so that $\sigma(t_1)|_\pi$ and $\sigma(t_2)|_\pi$ have different root symbols and $\pi$ is minimal with this property. If $\sigma(t_1)|_\pi$ is a function call, then the variables in this subterm do not count for $lvar(\sigma(t_1))$. If $\sigma(t_1)|_\pi$ is not a function call, then $\sigma(t_2)|_\pi$ must be a function call (otherwise these subterms have identical root symbols after successful unification) and $\sigma(t_2)|_\pi$ can be evaluated to $\sigma(t_1)|_\pi$ (in this case $\sigma(t_1)|_\pi$ is a ground term) or $\sigma(t_1)|_\pi = \sigma(t_2)|_\pi \in \rho$. In the latter case $lvar(\sigma(t_1)|_\pi) \subseteq var_\rho(\sigma(t_2)|_\pi)$ by definition of $var_\rho$. ■

Next we want to characterize the effect of a substitution with respect to the extension $var_\rho$ of a variable set.

**Lemma 4.6** *If $X \in var_\rho(t)$ and $\sigma$ is a substitution, then $lvar(\sigma(X)) \subseteq var_{\sigma(\rho)}(\sigma(t))$.*

*Proof:* Since $var_\rho(t) = var(t) \cup \{lvar(e) \mid f(\bar{t}) = e \in \rho \text{ with } var(\bar{t}) \subseteq var(t)\}$, there are two cases if $X \in var_\rho(t)$:

1. $X \in var(t)$: Then $lvar(\sigma(X)) \subseteq var(\sigma(X)) \subseteq var(\sigma(t)) \subseteq var_{\sigma(\rho)}(\sigma(t))$.
2. $X \in lvar(e)$ for some $f(\bar{t}) = e \in \rho$ with $var(\bar{t}) \subseteq var(t)$: Then $var(\sigma(\bar{t})) \subseteq var(\sigma(t))$. Hence $lvar(\sigma(X)) \subseteq lvar(\sigma(e)) \subseteq var_{\sigma(\rho)}(\sigma(t))$.

■

The following lemma extends the previous lemma to terms:

**Lemma 4.7** *If $t_1$ and $t_2$ are terms with $lvar(t_1) \subseteq var_\rho(t_2)$ and $\sigma$ is a substitution, then $lvar(\sigma(t_1)) \subseteq var_{\sigma(\rho)}(\sigma(t_2))$.*

*Proof:* Let $X \in lvar(\sigma(t_1))$. Then there is a variable $Y \in lvar(t_1)$ with $X \in lvar(\sigma(Y))$. Condition $lvar(t_1) \subseteq var_\rho(t_2)$ implies $Y \in var_\rho(t_2)$. By the previous lemma, $lvar(\sigma(Y)) \subseteq var_{\sigma(\rho)}(\sigma(t_2))$. Therefore $X \in var_{\sigma(\rho)}(\sigma(t_2))$. ■

Now we are able to prove the correctness of the abstract unification operation. Correctness means that abstract unification is *consistent* with concrete unification in the following sense:

> If $\langle \sigma, \rho \rangle \in \gamma(A)$ is the current substitution/residuation pair during program execution and the execution of the literal $t_1 = t_2$ yields the new substitution/residuation pair $\langle \sigma' \circ \sigma, \rho' \rangle$, then this new substitution/residuation pair is covered by the abstraction computed by abstract unification, i.e., $\langle \sigma' \circ \sigma, \rho' \rangle \in \gamma(amgu(A, t_1, t_2))$.

Execution of the literal $t_1 = t_2$ means applying the rules of Figure 2 to the equations $\rho \cup \{\sigma(t_1) = \sigma(t_2)\}$. In order to simplify the proof, we state a result for the unifier $\langle \overline{\sigma}, \overline{\rho} \rangle$ of the single equation $\{\sigma(t_1) = \sigma(t_2)\}$, i.e., we show $\langle \overline{\sigma} \circ \sigma, \overline{\rho} \cup \overline{\sigma}(\rho) \rangle \in \gamma(amgu(A, t_1, t_2))$. The difference between $\langle \overline{\sigma} \circ \sigma, \overline{\rho} \cup \overline{\sigma}(\rho) \rangle$ and $\langle \sigma' \circ \sigma, \rho' \rangle$ is that some variables in $\overline{\sigma}(\rho)$ are bound to ground terms by $\langle \sigma' \circ \sigma, \rho' \rangle$ (since the original computation considers also the residuations in $\rho$ which may be evaluated to ground constructor terms and thus binds variables of the other side of the equation) and $\rho'$ may contain less residuations than $\overline{\rho} \cup \overline{\sigma}(\rho)$. But this difference causes no problem since $\langle \overline{\sigma} \circ \sigma, \overline{\rho} \cup \overline{\sigma}(\rho) \rangle \in \gamma(A')$ implies $\langle \sigma' \circ \sigma, \rho' \rangle \in \gamma(A')$. This can be seen by the following two obvious propositions:

**Proposition 4.8** *If $A$ is an abstraction, $\langle \sigma, \rho \rangle \in \gamma(A)$ and $\sigma'$ a substitution which maps all variables $X$ with $\sigma(X) \neq X$ into ground constructor terms, then $\langle \sigma' \circ \sigma, \sigma'(\rho) \rangle \in \gamma(A)$.*

**Proposition 4.9** *If $A$ is an abstraction, $\langle \sigma, \rho \rangle \in \gamma(A)$ and $\rho' \subseteq \rho$ where the residuations from $\rho - \rho'$ do not contain variables, then $\langle \sigma, \rho' \rangle \in \gamma(A)$.*

Due to this argument we prove in the following correctness theorems all results w.r.t. unifiers which do not consider the current residuation $\rho$. The following theorem states the correctness of the abstract unification in this sense.

**Theorem 4.10 (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 $\sigma(t)$ computed by the rules of Figure 2, $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle \in \gamma(amgu(A, X, t))$.*

*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

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

We have to show: $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle \in \gamma(A')$.

1. $X \texttt{ if } \{Y\} \in A'$: Since $\langle \sigma', \rho' \rangle$ is a unifier for $\sigma(X)$ and $\sigma(Y)$, Lemma 4.5 yields

$$lvar(\sigma'(\sigma(X))) \subseteq var_{\rho'}(\sigma'(\sigma(Y))) \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(Y))).$$

2. $Y \texttt{ if } \{X\} \in A'$: Symmetric to the previous case.

3. $Z \texttt{ if } V \in A' \cap A$: Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $lvar(\sigma(Z)) \subseteq var_\rho(\sigma(V))$. The straightforward extension of Lemma 4.7 to sets of terms yields

$$lvar(\sigma'(\sigma(Z))) \subseteq var_{\sigma'(\rho)}(\sigma'(\sigma(V))) \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(V))).$$

   Hence all variable conditions of $A'$ are satisfied by $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle$.

4. $f(\bar{t})$ occurs directly in $\sigma'(\sigma(Z))$ with $var(\bar{t}) \neq \emptyset$ (for an arbitrary variable $Z$):

   First we assume that $f(\bar{t})$ occurs in a position also present in $\sigma(Z)$, i.e., there is a position $\pi$ with $\sigma'(\sigma(Z))|_\pi = f(\bar{t})$, $\sigma(Z)|_\pi = f(\bar{s})$ and $\sigma'(\bar{s}) = \bar{t}$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$ and $f(\bar{s})$ occurs directly in $\sigma(Z)$, $f \in A$ (which implies $f \in A'$) or $var(\bar{s}) \subseteq var(\sigma(V))$ for some $Z\,\texttt{with}\,f|_V \in A$. The latter case implies $var(\bar{t}) = var(\sigma'(\bar{s})) \subseteq var(\sigma'(\sigma(V)))$ for $Z\,\texttt{with}\,f|_V \in A'$.

   Otherwise we assume that $f(\bar{t})$ occurs directly in $\sigma'(\sigma(Z))$ but not in $\sigma(Z)$, i.e., there is a position $\pi$ with $\sigma'(\sigma(Z))|_\pi = f(\bar{t})$ but $\sigma(Z)|_\pi$ is undefined or a variable. In this case $\sigma(Z)$ must contain a variable which is instantiated by the unifier $\langle \sigma', \rho' \rangle$ to a non-constructor term. Since a unifier computed by the rules of Figure 2 binds only l-variables in $\sigma(X)$ and $\sigma(Y)$ to non-constructor terms, $\sigma(Z)$ must share a l-variable with $\sigma(X)$ or $\sigma(Y)$ (for simplicity we consider only the case for $\sigma(X)$ in the following), i.e., $\{Z, X\} \in A$ and $\sigma'(\sigma(X))$ have also a direct occurrence of the subterm $f(\bar{t})$. Since new function calls are not created by the rules of Figure 2, there must exist a subterm $f(\bar{s})$ of an instantiated variable $\sigma(Z')$ with $\sigma'(\bar{s}) = \bar{t}$. Moreover, $\sigma(Z')$ and $\sigma(X)$ (or $\sigma(Y)$) share a l-variable (which contains the function call $f(\bar{t})$ in $\sigma'$). Hence, because of $\langle \sigma, \rho \rangle \in \gamma(A)$, $\{Z', X\}$(or $\{Z', Y\}$) $\in A$ and $f \in A$ (which immediately implies $f \in A'$) or $var(\bar{s}) \subseteq var(\sigma(V))$ for some $Z'\,\texttt{with}\,f|_V \in A$. In the latter case we have $X\,\texttt{with}\,f|_V \in A'$ and also $Z\,\texttt{with}\,f|_V \in A'$ (since $A'$ is closed under the distribution of sharing information) where $var(\bar{t}) = var(\sigma'(\bar{s})) \subseteq var(\sigma'(\sigma(V)))$. Hence the occurrence of $f(\bar{t})$ in $\sigma'(\sigma(Z))$ is covered by $A'$.

5. $f(\bar{t})$ occurs directly in $\sigma'(\rho) \cup \rho'$ with $var(\bar{t}) \neq \emptyset$ but does not occur directly in $\sigma'(\sigma(Z))$ for any variable $Z$:

   - $f(\bar{t})$ belongs to $\sigma'(\rho)$: Since $f(\bar{t})$ does not occur in $\sigma'(\sigma(Z))$ for any $Z$, this function call is not contained in the image of $\sigma'$ and therefore there exists a residual function call $f(\bar{s})$ in $\rho$ with $\sigma'(\bar{s}) = \bar{t}$. Since $var(\bar{s}) \neq \emptyset$ and $\langle \sigma, \rho \rangle \in \gamma(A)$, $f \in A$ or $var(\bar{s}) \subseteq var(\sigma(V))$ for some $Z\,\texttt{with}\,f|_V \in A$. Thus $f \in A'$ or $var(\bar{t}) = var(\sigma'(\bar{s})) \subseteq var(\sigma'(\sigma(V)))$ for $Z\,\texttt{with}\,f|_V \in A'$.
   - $f(\bar{t})$ belongs to $\rho'$: Since $\rho'$ is generated during unification of $\sigma(X)$ and $\sigma(Y)$, there is a subterm $f(\bar{s})$ of $\sigma(X)$ or $\sigma(Y)$ with $\sigma'(\bar{s}) = \bar{t}$, i.e., $f(\bar{t})$ occurs in $\sigma'(\sigma(X))$ or $\sigma'(\sigma(Y))$ which contradicts our assumption. Thus this case cannot occur.

   Therefore every residual function call in $\sigma'(\rho) \cup \rho'$ is covered by $A'$.

6. $lvar(\sigma'(\sigma(Z))) \cap lvar(\sigma'(\sigma(Z'))) \neq \emptyset$ for variables $Z \neq Z'$:

   If $lvar(\sigma(Z)) \cap lvar(\sigma(Z')) \neq \emptyset$, then $\{Z, Z'\} \in A$ and thus $\{Z, Z'\} \in A'$. Otherwise we assume $lvar(\sigma(Z)) \cap lvar(\sigma(Z')) = \emptyset$. Since $\sigma'$ instantiates only l-variables of $\sigma(X)$ and $\sigma(Y)$ to non-ground terms, $\sigma(Z)$, $\sigma(Z')$ and $\sigma(X)$, $\sigma(Y)$ must share l-variables. Hence (in the worst case) $\{Z, X\}, \{Z', Y\} \in A$ which implies by definition of $A'$ (closure property) $\{Z, Z'\} \in A'$.

Altogether we have shown that $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle \in \gamma(A')$ for the case $t = Y$.

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

$$A := amgu(A, X, c(Y_1, \ldots, Y_n)) \;=\; closure(A \cup \{X\,\texttt{if}\,\{Y_1, \ldots, Y_n\},\ Y_1\,\texttt{if}\,\{X\}, \ldots,\ Y_n\,\texttt{if}\,\{X\},\ \{X, Y_1\}, \ldots,\ \{X, Y_n\}\})$$

1. $X$ `if` $\{Y_1, \ldots, Y_n\} \in A'$: Since $\langle \sigma', \rho' \rangle$ is a unifier for $\sigma(X)$ and $\sigma(t)$, Lemma 4.5 yields

$$lvar(\sigma'(\sigma(X))) \subseteq var_{\rho'}(\sigma'(\sigma(t))) \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(t))) = var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(\{Y_1, \ldots, Y_n\}))).$$

2. $Y_i$ `if` $\{X\} \in A'$: Similarly to the previous case, Lemma 4.5 yields $lvar(\sigma'(\sigma(t))) \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(X)))$. Since $lvar(\sigma'(\sigma(Y_i))) \subseteq lvar(\sigma'(\sigma(t)))$, we obtain

$$lvar(\sigma'(\sigma(Y_i))) \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(X))).$$

3. $Z$ `if` $V \in A' \cap A$: This is identical to the corresponding case of "$t = Y$" (see above).

   Therefore all variable conditions of $A'$ are satisfied by $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle$.

4. $f(\bar{t})$ occurs directly in $\sigma'(\sigma(Z))$ or in the residuation $\sigma'(\rho) \cup \rho'$ with $var(\bar{t}) \neq \emptyset$: This is similar to the corresponding cases of "$t = Y$" with the difference that sometimes we have to replace "$Y$" by "some $Y_i$" in the proof.

5. $lvar(\sigma'(\sigma(Z))) \cap lvar(\sigma'(\sigma(Z'))) \neq \emptyset$ for variables $Z \neq Z'$: Let $lvar(\sigma(Z)) \cap lvar(\sigma(Z')) = \emptyset$ (otherwise we proceed as in case "$t = Y$"). Since $\sigma'$ instantiates only l-variables of $\sigma(X)$ and $\sigma(c(Y_1, \ldots, Y_n))$ to non-ground terms, $\sigma(Z)$ and $\sigma(Z')$ must share l-variables with $\sigma(X)$ and $\sigma(c(Y_1, \ldots, Y_n))$. By $\langle \sigma, \rho \rangle \in \gamma(A)$ and the closure property of $A'$ we obtain $\{Z, Z'\} \in A'$.

Hence we have proved the theorem for the case $t = c(Y_1, \ldots, Y_n)$.
Next we consider the case $t = f(Y_1, \ldots, Y_n)$. Then

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

1. $X$ `if` $\{Y_1, \ldots, Y_n\} \in A'$:

   If $\sigma(t)$ is a ground term, then the function call $\sigma(t)$ evaluates to a ground term which is unified with $\sigma(X)$, i.e., all l-variables of $\sigma(X)$ are bound to ground terms. Hence $lvar(\sigma'(\sigma(X))) = \emptyset$.

   If $\sigma(t)$ is not a ground term, then the function call $\sigma(t)$ delays or is bound to $\sigma(X)$, i.e., there are the following two cases:

   - If $\sigma(X)$ is not a variable, then $\sigma' = \emptyset$ and $\rho' = \{\sigma(X) = \sigma(t)\}$. Hence $lvar(\sigma'(\sigma(X))) = lvar(\sigma(X)) \subseteq var_{\rho'}(\sigma(t)) \subseteq var_{\rho' \cup \rho}(\sigma(t)) = var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(\{Y_1, \ldots, Y_n\})))$.
   - If $\sigma(X)$ is a variable, then $\sigma' = \{\sigma(X) \mapsto \sigma(t)\}$ and $\rho' = \emptyset$. Hence $lvar(\sigma'(\sigma(X))) = lvar(\sigma(t)) = \emptyset \subseteq var_{\rho' \cup \sigma'(\rho)}(\sigma'(\sigma(\{Y_1, \ldots, Y_n\})))$.

2. $Z$ `if` $V \in A' \cap A$: This is identical to the corresponding case of "$t = Y$" (see above).

3. $f(\bar{t})$ occurs directly in $\sigma'(\sigma(Z))$ or in the residuation $\sigma'(\rho) \cup \rho'$ with $var(\bar{t}) \neq \emptyset$: We assume that $f(\bar{t})$ is a "new" residual function call introduced by this unification, i.e., $f(\bar{t}) = \sigma'(\sigma(f(Y_1, \ldots, Y_n)))$ (otherwise we proceed as in case "$t = Y$"). But this function call is covered by $A'$ since $X$ `with` $f|_{\{Y_1, \ldots, Y_n\}} \in A'$, $var(\bar{t}) = var(\sigma'(\sigma(\{Y_1, \ldots, Y_n\})))$ and the closure property of $A'$.

4. $lvar(\sigma'(\sigma(Z))) \cap lvar(\sigma'(\sigma(Z'))) \neq \emptyset$ for variables $Z \neq Z'$: Let $lvar(\sigma(Z)) \cap lvar(\sigma(Z')) = \emptyset$ (otherwise we proceed as in case "$t = Y$"). Then $\sigma(X)$ must be a variable and thus $\sigma' = \{\sigma(X) \mapsto \sigma(t)\}$. Therefore $\sigma'$ may only *delete* occurrences of $\sigma(X)$ from the l-variables but does not add new l-variables in any term, i.e., $lvar(\sigma'(\sigma(Z))) \cap lvar(\sigma'(\sigma(Z'))) = \emptyset$. By this contradiction we infer that this case cannot occur.

■

Next we want to prove that the abstract operations performed at the entry of a clause are correct w.r.t. the concrete semantics. Hence we must show something like:

> If $P$ is a predicate call with abstraction $A$, $\langle \sigma, \rho \rangle \in \gamma(A)$, $L \texttt{ :- } L1, \ldots, L_k$ is a variant of a clause and $\langle \sigma', \rho' \rangle$ is a unifier for $\sigma(P)$ and $L$, then $\langle \sigma' \circ \sigma, \rho' \cup \sigma'(\rho) \rangle \in \gamma(A_0)$ where $A_0$ is the abstraction computed for the clause $L \texttt{ :- } L1, \ldots, L_k$ by the abstract interpretation algorithm *ai*.

However, this statement is too strong and not true in general since $A_0$ contains only properties of variables occurring in the clause $L \texttt{ :- } L_1, \ldots, L_k$. Bruynooghe [Bru91] has shown that it is sufficient to prove that the call substitution *restricted to the clause variables* is contained in the computed call abstraction. Hence in our case it is sufficient that $\langle \sigma' \circ \sigma|_W, \rho' \rangle \in \gamma(A_0)$ where $W$ is the set of all clause variables and the restriction of a substitution $\phi$ to a variable set $V$ is defined by

$$\phi|_V := \{X \mapsto t \ \in \phi \mid X \in V\}$$

Strictly speaking we have omitted the "old" residuation $\sigma'(\rho)$ while applying the clause, but later we will see that this omitted residuation set is also covered by the computed success abstraction (compare Theorem 4.14). The difference of this simplification in comparison to the "real" computation is that some residuations of $\sigma'(\rho)$ may be evaluated during the concrete clause application. But this difference makes no problem due to Propositions 4.8 and 4.9.

**Theorem 4.11 (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 $p(Z_1, \ldots, Z_n) \texttt{ :- } L_1, \ldots, L_k$ be a (renamed) clause, $\langle \sigma', \rho' \rangle$ be a unifier for $\sigma(P)$ and $p(Z_1, \ldots, Z_n)$ computed by the rules of Figure 2, and $A_0$ be the abstraction computed by algorithm ai. Then $\langle \sigma' \circ \sigma|_W, \rho' \rangle \in \gamma(A_0)$ with $W = var(p(Z_1, \ldots, Z_n) \texttt{ :- } L_1, \ldots, L_k)$.*

*Proof:* $\sigma' = \{Z_1 \mapsto \sigma(X_1), \ldots, Z_n \mapsto \sigma(X_n)\}$ and $\rho' = \emptyset$ is a unifier computed for $\sigma(P)$ and $p(Z_1, \ldots, Z_n)$ (all other unifiers are renamings of this). Since all $Z \in W$ are new variables, $\sigma(Z) = Z$ and thus $\sigma' \circ \sigma|_W = \sigma'$. Hence we have to show: $\langle \sigma', \emptyset \rangle \in \gamma(A_0)$.

1. $X \texttt{ if } V \in A_0$: By definition of *call_restrict* and *ai*, $V = \emptyset$, $X = Z_i$ for some $i \in \{1, \ldots, n\}$ and $X_i \in A$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $lvar(\sigma'(Z_i)) = lvar(\sigma(X_i)) = \emptyset$. Hence all variable conditions of $A_0$ are satisfied by $\langle \sigma', \emptyset \rangle$.

2. $f(\bar{t})$ occurs directly in $\sigma'(X)$ with $var(\bar{t}) \neq \emptyset$: Then $X = Z_i$ for some $i \in \{1, \ldots, n\}$ and therefore $f(\bar{t})$ occurs directly in $\sigma(X_i)$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $f \in A$ (which immediately implies $f \in A_0$) or $var(\bar{t}) \subseteq var(\sigma(V))$ for some $X_i \texttt{ with } f|_V \in A$. Now consider the latter

case. If $V \not\subseteq \{X_1, \ldots, X_n\}$, then $f \in A_0$ by definition of *call_restrict*. Otherwise, if $V \subseteq \{X_1, \ldots, X_n\}$, $Z_i \,\texttt{with}\, f|_{\sigma_{xz}(V)} \in A_0$ (where the substitution $\sigma_{xz}$ renames each $X_j$ into $Z_j$). Since $\sigma_{xz}(V) \subseteq \{Z_1, \ldots, Z_n\}$, $var(\overline{t}) \subseteq var(\sigma(V)) = var(\sigma'(\sigma_{xz}(V)))$.

3. Since the residuation part is empty, there are no function calls in this part.

4. $lvar(\sigma'(X)) \cap lvar(\sigma'(Y)) \neq \emptyset$ for variables $X \neq Y$: Then $X = Z_i$ and $Y = Z_j$ for some $i \neq j$. $\sigma'(Z_i) = \sigma(X_i)$ and $\sigma'(Z_j) = \sigma(X_j)$ implies $lvar(\sigma(X_i)) \cap lvar(\sigma(X_j)) \neq \emptyset$. Since $\langle \sigma, \rho \rangle \in \gamma(A)$, $\{X_i, X_j\} \in A$ and hence $\{Z_i, Z_j\} \in A_0$.

■

To prove a similar theorem for the correctness of the abstract operations performed at clause exit, we need two propositions about the flow of residuation abstractions in the abstract interpretation of predicates. The first proposition states that a residuation abstraction of the form "$f$" will never be deleted during abstract interpretation:

**Proposition 4.12** *Let $P$ be a predicate call with abstraction $A$, $f \in A$ and $ai(A, P) \neq \bot$. Then $f \in ai(A, P)$.*

*Proof:* By induction on the computation steps of the abstract interpretation algorithm (cf. Section 4.3), it is straightforward to show that this proposition holds since an abstraction element "$f$" is passed through abstract unification (*amgu*), from predicate calls to clause entries (*call_restrict*) and from clause exits to predicate calls (*exit_restrict* and *rest*). Hence "$f$" is present in all abstractions different from $\bot$ during the entire abstract interpretation of $ai(A, P)$. ■

The next proposition states that a residuation abstraction of the form "$X \,\texttt{with}\, f|_V$" will only be deleted during abstract interpretation if all variables are provable bound to ground terms:

**Proposition 4.13** *Let $P$ be a predicate call with abstraction $A$, $X \,\texttt{with}\, f|_V \in A$ and $A' := ai(A, P) \neq \bot$. Then $f \in A'$ or $V = V_1 \cup V_2$ with $X \,\texttt{with}\, f|_{V_1} \in A'$ and all variables $Z \in V_2$ are bound to ground terms in all concrete interpretations corresponding to $A'$.*

*Proof:* This proposition holds similarly to the previous proposition. Note that an abstraction element $X \,\texttt{with}\, f|_V$ is never deleted but only transformed into $f$ (by *call_restrict* or *exit_restrict*) or some variables in $V$ are deleted by the normalization rules. In the latter case the conditions in the normalization rules ensure that these deleted variables are bound to ground terms in the corresponding concrete interpretations (see proof of Lemma 4.1). ■

Now we are prepared to 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 4.14 (Correctness of clause exit)** *Let $P = p(X_1, \ldots, X_n)$ be a predicate call with abstraction $A_{in}$ and $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$. Let $A = ai(A_{in}, P) = closure(A_{success} \cup rest(A_{in}, \{X_1, \ldots, X_n\}))$ 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. Let* $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$ *and* $\sigma$ *be a substitution on the variables from* $\sigma_{in}(P)$ *so that* $\sigma(\sigma_{in}(P)) = \sigma_k(L)$*. Then* $\langle \sigma \circ \sigma_{in}, \rho_k \cup \sigma(\rho_{in}) \rangle \in \gamma(A)$*.*

In a concrete computation the substitution/residuation pair $\langle \sigma_k, \rho_k \rangle$ after the clause application is an extension of $\sigma_{in}$ (i.e., $\sigma_k(L) = \sigma(\sigma_{in}(P))$, as required) and an extension of $\rho_{in}$ (i.e., $\rho_k = \rho \cup \sigma(\rho_{in})$ for some $\rho$ where some of the residuations can be evaluated). Since we have omitted the residuation $\rho_{in}$ in the corresponding clause entry (compare Theorem 4.11), we prove in this theorem that this was correct.

*Proof:* Let $L = p(Z_1, \ldots, Z_n)$, $\sigma_{zx} = \{Z_1 \mapsto X_1, \ldots, Z_n \mapsto X_n\}$ be the renaming of each variable $Z_i$ into $X_i$, and $\sigma_{xz} = \{X_1 \mapsto Z_1, \ldots, X_n \mapsto Z_n\}$ the inverse of $\sigma_{zx}$. Note that $\sigma_k(Z_i) = \sigma(\sigma_{in}(X_i))$ for $i = 1, \ldots, n$. We have to show: $\langle \sigma \circ \sigma_{in}, \rho_k \cup \sigma(\rho_{in}) \rangle \in \gamma(A)$.

1. $X \,\mathtt{if}\, V \in A$: Hence there are two cases:

   - $X \,\mathtt{if}\, V \in \mathit{rest}(A_{in}, \{X_1, \ldots, X_n\})$: Then $\mathit{lvar}(\sigma_{in}(X)) \subseteq \mathit{var}_{\rho_{in}}(\sigma_{in}(V))$ since $X \,\mathtt{if}\, V \in A_{in}$ and $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$. Lemma 4.7 yields $\mathit{lvar}(\sigma(\sigma_{in}(X))) \subseteq \mathit{var}_{\sigma(\rho_{in})}(\sigma(\sigma_{in}(V))) \subseteq \mathit{var}_{\rho_k \cup \sigma(\rho_{in})}(\sigma(\sigma_{in}(V)))$.
   - $X \,\mathtt{if}\, V \in A_{success}$: Since $A_{exit} \sqsubseteq A_{success}$, there is a set $V' \subseteq V$ with $X \,\mathtt{if}\, V' \in A_{exit}$. By definition of $A_{exit}$, $\sigma_{xz}(X) \,\mathtt{if}\, \sigma_{xz}(V') \in A_k$ and $\{\sigma_{xz}(X)\} \cup \sigma_{xz}(V') \subseteq \{Z_1, \ldots, Z_n\}$. $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$ implies $\mathit{lvar}(\sigma_k(\sigma_{xz}(X))) \subseteq \mathit{var}_{\rho_k}(\sigma_k(\sigma_{xz}(V')))$ and hence (since $\sigma(\sigma_{in}(P)) = \sigma_k(L)$) $\mathit{lvar}(\sigma(\sigma_{in}(X))) \subseteq \mathit{var}_{\rho_k}(\sigma(\sigma_{in}(V')))$. Therefore $\mathit{lvar}(\sigma(\sigma_{in}(X))) \subseteq \mathit{var}_{\rho_k}(\sigma(\sigma_{in}(V))) \subseteq \mathit{var}_{\rho_k \cup \sigma(\rho_{in})}(\sigma(\sigma_{in}(V)))$.

2. $f(\bar{t})$ occurs directly in $\sigma(\sigma_{in}(X_i))$ (for some $i \in \{1, \ldots, n\}$) with $\mathit{var}(\bar{t}) \neq \emptyset$: Then $f(\bar{t})$ occurs also directly in $\sigma_k(Z_i)$ since $\sigma(\sigma_{in}(P)) = \sigma_k(L)$. $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$ implies $f \in A_k$ (which immediately implies $f \in A$) or $\mathit{var}(\bar{t}) \subseteq \mathit{var}(\sigma_k(V))$ for some $Z_i \,\mathtt{with}\, f|_V \in A_k$. In the latter case there are two possibilities: If $V \not\subseteq \{Z_1, \ldots, Z_n\}$, then $f \in A_{out}$ (by definition of *exit_restrict*) and $f \in A$. If $V \subseteq \{Z_1, \ldots, Z_n\}$, then $X_i \,\mathtt{with}\, f|_{\sigma_{zx}(V)} \in A_{exit}$. This implies $X_i \,\mathtt{with}\, f|_{\sigma_{zx}(V)} \in A$ where $\mathit{var}(\bar{t}) \subseteq \mathit{var}(\sigma_k(V)) = \mathit{var}(\sigma(\sigma_{in}(\sigma_{zx}(V))))$.

3. $f(\bar{t})$ occurs directly in $\sigma(\sigma_{in}(X))$ with $X \notin \{X_1, \ldots, X_n\}$ and $\mathit{var}(\bar{t}) \neq \emptyset$: Hence there is a position $\pi$ with $f(\bar{t}) = \sigma(\sigma_{in}(X))|_\pi$. We can distinguish two cases:

   - $\pi$ is also a position in $\sigma_{in}(X)$ and $\sigma_{in}(X)|_\pi = f(\bar{s})$ with $\sigma(\bar{s}) = \bar{t}$: Since $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$, $f \in A_{in}$ (which implies $f \in A$ by Proposition 4.12) or $\mathit{var}(\bar{s}) \subseteq \mathit{var}(\sigma_{in}(V))$ for some $X \,\mathtt{with}\, f|_V \in A_{in}$. Since $X \notin \{X_1, \ldots, X_n\}$, the latter case yields $X \,\mathtt{with}\, f|_V \in \mathit{rest}(A_{in}, \{X_1, \ldots, X_n\})$ and thus $X \,\mathtt{with}\, f|_V \in A$ where $\mathit{var}(\bar{t}) = \mathit{var}(\sigma(\bar{s})) \subseteq \mathit{var}(\sigma(\sigma_{in}(V)))$.
   - $\pi$ is not a position in $\sigma_{in}(X)$ or $\sigma_{in}(X)|_\pi$ is a variable: Then $\sigma_{in}(X)$ contains a variable $Z$ which is instantiated by $\sigma$ to a term containing the subterm $f(\bar{t})$. Since $\sigma$ instantiates only variables occurring in $\sigma_{in}(P)$, there is a variable $X_i$ so that $\sigma_{in}(X_i)$ has a direct occurrence of $Z$. Hence $Z$ is shared between $\sigma_{in}(X_i)$ and $\sigma_{in}(X)$ which implies $\{X_i, X\} \in A_{in}$, $\{X_i, X\} \in \mathit{rest}(A_{in}, \{X_1, \ldots, X_n\})$, and also $\{X_i, X\} \in A$ (since $X \notin \{X_1, \ldots, X_n\}$). Moreover, $f(\bar{t})$ occurs directly in $\sigma(\sigma_{in}(X_i))$. Hence we obtain as

above $f \in A$ or $var(\bar{t}) \subseteq var(\sigma(\sigma_{in}(V))$ for some $X_i \,\texttt{with}\, f|_V \in A$. The latter case implies $X \,\texttt{with}\, f|_V \in A$ since $\{X_i, X\} \in A$ and $A$ is closed under the rule for distribution of sharing information.

4. $f(\bar{t})$ with $var(\bar{t}) \neq \emptyset$ occurs directly in $\rho_k$ but not directly in $\sigma(\sigma_{in}(X))$ for all variables $X$ (otherwise proceed as in case 2 or 3): Since $\langle \sigma_k, \rho_k \rangle \in \gamma(A_k)$, $f \in A_k$ (which immediately implies $f \in A$) or $var(\bar{t}) \subseteq var(\sigma_k(V))$ for some $X \,\texttt{with}\, f|_V \in A_k$. In the latter case there are two possibilities: If $\{X\} \cup V \not\subseteq \{Z_1, \ldots, Z_n\}$, then $f \in A_{out}$ (by definition of *exit_restrict*) and hence $f \in A$. If $\{X\} \cup V \subseteq \{Z_1, \ldots, Z_n\}$, then $\sigma_{zx}(X) \,\texttt{with}\, f|_{\sigma_{zx}(V)} \in A_{exit}$. This implies $\sigma_{zx}(X) \,\texttt{with}\, f|_{\sigma_{zx}(V)} \in A$ where $var(\bar{t}) \subseteq var(\sigma_k(V)) = var(\sigma(\sigma_{in}(\sigma_{zx}(V))))$.

5. $f(\bar{t})$ with $var(\bar{t}) \neq \emptyset$ occurs directly in $\sigma(\rho_{in})$ but not directly in $\sigma(\sigma_{in}(X))$ for all variables $X$ (otherwise proceed as in case 2 or 3): Since $f(\bar{t})$ does not occur directly in any $\sigma(\sigma_{in}(X))$, $f(\bar{s})$ occurs directly in $\rho_{in}$ with $\sigma(\bar{s}) = \bar{t}$. Since $\langle \sigma_{in}, \rho_{in} \rangle \in \gamma(A_{in})$, $f \in A_{in}$ (which implies $f \in A$ by Proposition 4.12) or $var(\bar{s}) \subseteq var(\sigma_{in}(V))$ for some $X \,\texttt{with}\, f|_V \in A_{in}$. In the latter case there are three possibilities:

   - $X \notin \{X_1, \ldots, X_n\}$: Then $X \,\texttt{with}\, f|_V \in rest(A_{in}, \{X_1, \ldots, X_n\})$ and hence $X \,\texttt{with}\, f|_V \in A$ where $var(\bar{t}) = var(\sigma(\bar{s})) \subseteq var(\sigma(\sigma_{in}(V))$.
   - $X \in \{X_1, \ldots, X_n\}$, $V \not\subseteq \{X_1, \ldots, X_n\}$: Then $f \in A_{call}$ (by definition of *call_restrict*) and $f \in A$ (by Proposition 4.12).
   - $\{X\} \cup V \subseteq \{X_1, \ldots, X_n\}$: Then, by Proposition 4.13, $f \in A$ or $V = V_1 \cup V_2$ with $X \,\texttt{with}\, f|_{V_1} \in A$ and all $Z \in V_2$ are bound to ground terms by $\sigma \circ \sigma_{in}$. The latter case implies $var(\bar{t}) = var(\sigma(\bar{s})) \subseteq var(\sigma(\sigma_{in}(V)) = var(\sigma(\sigma_{in}(V_1))$.

6. $lvar(\sigma(\sigma_{in}(X)) \cap lvar(\sigma(\sigma_{in}(Y))) \neq \emptyset$ for variables $X \neq Y$. We distinguish the following cases for the variables $X$ and $Y$:

   - $X, Y \in \{X_1, \ldots, X_n\}$: Since $\sigma(\sigma_{in}(P)) = \sigma_k(L)$, $lvar(\sigma_k(\sigma_{xz}(X)) \cap lvar(\sigma_k(\sigma_{xz}(Y)) \neq \emptyset$ which implies $\{\sigma_{xz}(X), \sigma_{xz}(Y)\} \in A_k$. Hence $\{X, Y\} \in A_{success}$ and $\{X, Y\} \in A$.
   - $X \notin \{X_1, \ldots, X_n\}$ (the case $Y \notin \{X_1, \ldots, X_n\}$ is symmetric and therefore omitted) and $lvar(\sigma_{in}(X)) \cap lvar(\sigma_{in}(Y)) \neq \emptyset$: Then $\{X, Y\} \in A_{in}$ which yields $\{X, Y\} \in rest(A_{in}, \{X_1, \ldots, X_n\})$ and $\{X, Y\} \in A$.
   - $X \notin \{X_1, \ldots, X_n\}$ and $lvar(\sigma_{in}(X)) \cap lvar(\sigma_{in}(Y)) = \emptyset$:

     First suppose $Y \in \{X_1, \ldots, X_n\}$. Since $\sigma$ does only instantiate variables from $\sigma_{in}(P)$, $\sigma_{in}(X)$ must share a l-variable with some $\sigma_{in}(X_i)$, and $\sigma(\sigma_{in}(X_i))$ shares another l-variable with $\sigma(\sigma_{in}(Y))$. Hence $\{X, X_i\} \in A_{in}$ which implies $\{X, X_i\} \in rest(A_{in}, \{X_1, \ldots, X_n\})$, and $\{X_i, Y\} \in A_{success}$ (as in the first case). Both facts imply $\{X, X_i\} \in A$ and $\{X_i, Y\} \in A$. Since $A$ is closed under the rule for transitivity of sharing information, $\{X, Y\} \in A$.

     Now suppose $Y \notin \{X_1, \ldots, X_n\}$. Since $\sigma$ does only instantiate variables from $\sigma_{in}(P)$, $\sigma_{in}(X)$ must share a l-variable with some $\sigma_{in}(X_i)$ and $\sigma_{in}(Y)$ must share a l-variable with some $\sigma_{in}(X_j)$ so that $\sigma(\sigma_{in}(X_i))$ shares another l-variable with $\sigma(\sigma_{in}(X_j))$. Hence $\{X, X_i\}, \{Y, X_j\} \in A_{in}$ which implies $\{X, X_i\}, \{Y, X_j\} \in rest(A_{in}, \{X_1, \ldots, X_n\})$. If

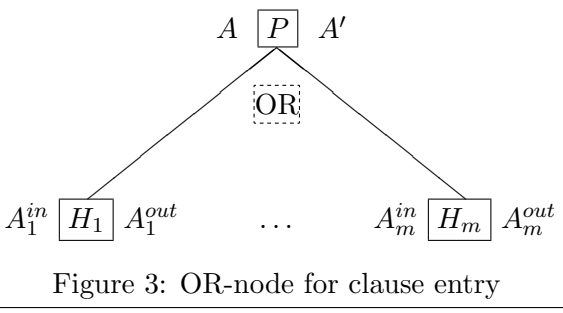


Figure 3: OR-node for clause entry

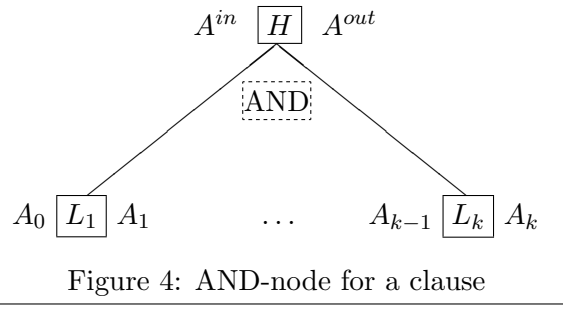


Figure 4: AND-node for a clause

$i = j$, then $\{X, Y\} \in A$ since $A$ is a closed abstraction. If $i \neq j$, then $\{X_i, X_j\} \in A_{success}$ and $\{X_i, X_j\} \in A$ (as in the first case). But this implies $\{X, Y\} \in A$ by the closure property of $A$.

■

### 4.3 Correctness of the abstract interpretation algorithm

Until now 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 [Bru91]. 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 (or a variant of) a predicate call in an ancestor node, then this call node is identified with the 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 3 ($H_1, \ldots, H_m$

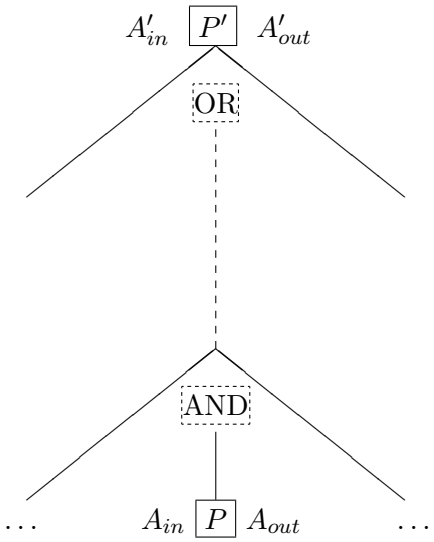


Figure 5: Recursive call: $P$ is a renaming of $P'$ and $A_{in}$ restricted to call $P$ is a renaming of $A'_{in}$ restricted to call $P'$

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 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 remainder of the 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 5): Then the success abstraction of $P'$ ($A'_{out}$ without the remainder of $A'_{in}$, i.e., $A_{success}$ in algorithm $ai$ in Section 3.2) is taken as the success abstraction of $P$ (or $\bot$ if it is not available). The combination of this success abstraction with the remainder of $A_{in}$ 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 [Bru91] 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 satisfies certain correctness conditions. Theorems 4.10, 4.11 and 4.14 imply exactly these correctness conditions. Hence we can infer the correctness of our abstract interpretation algorithm.

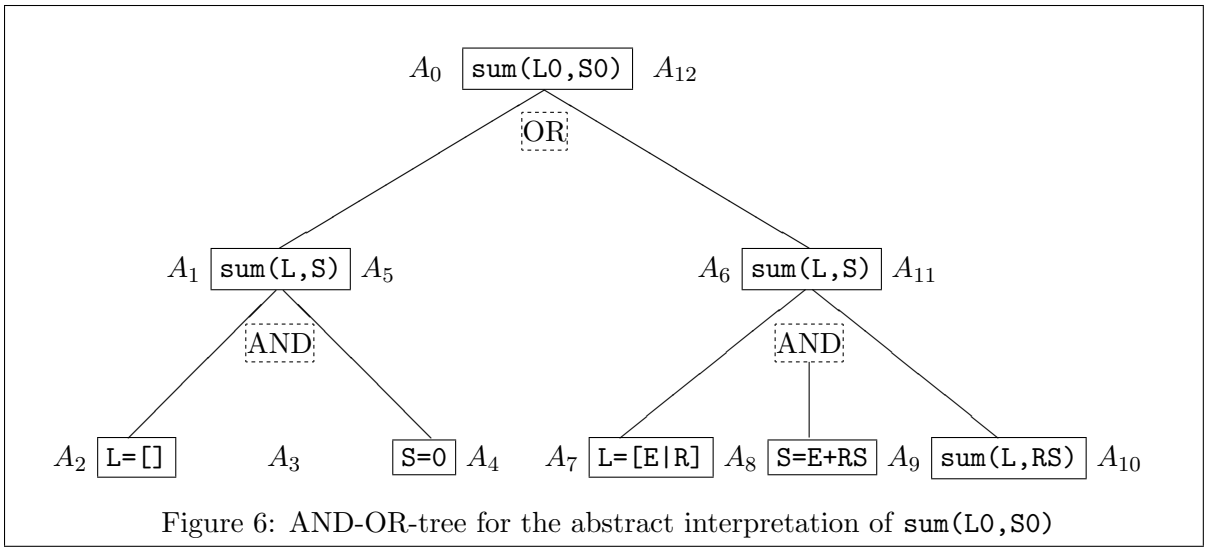


Figure 6: AND-OR-tree for the abstract interpretation of `sum(L0,S0)`

### 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:

| Goal: | Current residuation: | Current substitution: |
|---|---|---|
| `sum([1,3,5],S)` | $\emptyset$ | $\emptyset$ |
| `sum([3,5],RS1)` | {`1+RS1=S`} | $\emptyset$ |
| `sum([5],RS2)` | {`1+RS1=S, 3+RS2=RS1`} | $\emptyset$ |
| `sum([],RS3)` | {`1+RS1=S, 3+RS2=RS1, 5+RS3=RS2`} | $\emptyset$ |
| $\emptyset$ | $\emptyset$ | {`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 6. 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.

- $A_1 = \{\texttt{L}\}$ and $A_6 = \{\texttt{L}\}$: 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 is computed from $A_0$ by *call_restrict* and renaming.

- $A_2 = \{\texttt{L}\}$: The entry abstraction of the clause is also the abstraction for the first predicate call in the clause body.

- $A_3 = \{\texttt{L}\}$: The abstraction $A_2$ is not modified by abstract unification since `L` is already ground.

- $A_4 = \{\texttt{L}, \texttt{S}\}$: `S` is added to the abstraction by abstract unification since it is bound to a ground term after this unification.

- $A_5 = \{\texttt{L}, \texttt{S}\}$: The exit abstraction of this clause is the exit abstraction of the last body literal restricted to the variables in the clause head.

- $A_7 = \{\texttt{L}\}$: The entry abstraction of the second clause is also the abstraction for the first predicate call in the clause body.

- $A_8 = \{\texttt{L}, \texttt{E}, \texttt{R}\}$: The variables `E` and `R` are ground since `L` is ground. This is computed by the abstract unification algorithm together with the normalization rules.

- $A_9 = \{\texttt{L}, \texttt{E}, \texttt{R}, \texttt{S}\,\texttt{if}\,\{\texttt{RS}\}, \texttt{S}\,\texttt{with}\,\texttt{+}|_{\{\texttt{RS}\}}\}$: The function call to `+` is added to the abstraction. It can not be evaluated until the variable `RS` is ground.

- $A_{10} = \bot$: The call abstraction part of $A_9$ is $\{\texttt{L}\}$ (compare definition of *call_restrict*). Hence this predicate call is a renaming of the predicate call at the root and therefore we take the value $\bot$ 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 $\bot$, we start a recomputation at this point.

- $A_{11} = \bot$: The exit abstraction of the second clause is the exit abstraction of the last body literal.

- $A_{12} = \{\texttt{L0}, \texttt{S0}\}$: The success abstraction of the root predicate call is the least upper bound of $\{\texttt{L0}, \texttt{S0}\}$ and $\bot$ together with the remainder of $A_0$ (which is actually empty). Since the success abstraction of the root call is now available and different from $\bot$, we restart the evaluation of the abstraction $A_{10}$.

- $A_{10} = \{\texttt{L}, \texttt{RS}, \texttt{E}, \texttt{R}, \texttt{S}\}$: The new value of $A_{10}$ is computed from the new renamed success abstraction of the root predicate call ($\{\texttt{L}, \texttt{RS}\}$) together with the remainder of $A_9$ giving $\{\texttt{L}, \texttt{RS}, \texttt{E}, \texttt{R}, \texttt{S}\,\texttt{if}\,\{\texttt{RS}\}, \texttt{S}\,\texttt{with}\,\texttt{+}|_{\{\texttt{RS}\}}\}$. This abstraction simplified by the normalization rules is the new value of $A_{10}$.

- $A_{11} = \{\texttt{L}, \texttt{S}\}$: The exit abstraction of the second clause is the exit abstraction of the last body literal restricted to the variables in the clause head.

- $A_{12} = \{$`L0`, `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) together with the remainder of $A_0$ (which is actually empty). 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 has computed the exit abstraction {`L0`, `S0`} for the initial goal, we conclude by the correctness of the abstract interpretation algorithm and the concretisation function $\gamma$ that variable `S0` is bound to a ground term without 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-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 [GM86], SLOG [Fri85], BABEL [MR92], or ALF [Han90]) 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. 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.

Marriott, Søndergaard and Dart [MSD90] 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 is a simpler problem than our analysis of residuating logic programs due to the following reasons:

1. In their context only entire literals can be delayed and not single subterms. Therefore in their framework it is not necessary to analyse the precise structure of the terms.

2. 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 analyse the data flow in this case.

3. 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 side, we cannot analyse 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. But the evaluation of a negated literal may cause the evaluation of other negated literals and therefore it is not an atomic operation. However, 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 [Han90]).

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 the floundering problem of NU-Prolog or to derive run-time properties of the Andorra computation rule [HB88].

# References


[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AK90] 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.

[AKLN87] 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.

[BGL+87] 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.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, third rev. and ext. edition, 1987.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[Fri85] 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.

[GM86] 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.

[Han90] 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.

[Han91] M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.

[HB88] 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.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[MM82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

[MR92] 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.

[MSD90] 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.

[Nai91] 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.

[Nil90] 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.

[Smo91] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Research Report 12, DEC Paris Research Laboratory, 1991.

[SY86] 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.