Abstract Interpretation for (Constraint) Logic Programming*

Maurice Bruynooghe Dmitri Boulanger

Report CW183, November 1993

Department of Computing Science, K.U. Leuven

Abstract

The theory of abstract interpretation, initially developed by the Cousot's for imperative languages has been adapted to logic programming where it has become a very active area of research. It has reached a level of maturity where it is included as an optimisation tool in some prototype implementations of Prolog. In recent years, also abstract interpretation of constraint logic programming has been studied. Abstract interpretation is an area where the majority of publications is very technical in nature. This tutorial aims at making the topic more accessible to newcomers. Therefore, we have tried to present the main intuitions in a rather informal way. We have preferred informal accounts above very precise but rigid and hard to penetrate formalism. The paper is not a survey, so although we give some entry points to the literature, we have not aimed at completeness.

A first part discusses the framework of abstract interpretation in very general terms, as a way to approximate the semantics which has to be formulated as a fixpoint of some function. A second part applies this theory to logic programming, at the same time showing general examples of the theory, and explaining some of the major approaches towards abstract interpretation of logic programs. A third and a final part tries to explain that the step from logic programming to constraint logic programming is a small one and that frameworks developed for abstract interpretation of logic programs carry over to constraint logic programming.

1. Abstract Interpretation for (Constraint) Logic Programming

The theory of abstract interpretation, initially developed by the Cousot's for imperative languages has been adapted to logic programming where it has become a very active area of research, illustrated by the list of references in the survey paper [CoC92b]. It has reached a level of maturity where it is included as an optimisation tool in some prototype implementations of Prolog. In recent years, also abstract interpretation of constraint logic programming has been studied. Abstract interpretation is an area where the majority of publications is very technical in nature. This tutorial aims at making the topic more accessible to newcomers. Therefore, we have tried to present the main intuitions in a rather informal way. We have preferred informal accounts above very precise but rigid and hard to penetrate formalism. The paper is not a survey, so although we give some entry points to the literature, we have not aimed at completeness.

A first part discusses the framework of abstract interpretation in very general terms, as a way to approximate the semantics which has to be formulated as a fixpoint of some function. A second part applies this theory to logic programming, at the same time showing general examples of the theory, and explaining some of the major approaches towards abstract interpretation of logic programs. A third and a final part tries to explain that the step from logic programming to constraint logic programming is a small one and that frameworks developed for abstract interpretation of logic programs carry over to constraint logic programming.

This chapter assumes some familiarity with the basic notions of logic programming and with the concept of fixpoint. The reader can find material concerning these in [Llo87].

1.1 Basic Principles

1.1.1 Abstracting computations

As part of our education we have learned to compute properties of the results of calculations by performing approximate or abstract computations. The general idea is to abstract the operands, apply an abstract operator and to obtain a result which is the abstraction of the result of the concrete operation.

*~~Revised~~ Constraint Programming, Eds. B.Mayoh, E.Tóugu, J.Penjam, NATO Advanced Science Series, Computers and System Sciences, Springer-Verlag, 1994

A typical example mastered in the first years of school, is the multiplication of multidigit numbers. To check against silly errors, due to a slip of our mind, we have learned to perform an independent abstract computation: take the operands modulo 9¹ and multiply them modulo 9. The result should be equal to the result, modulo 9, of the original multiplication. Although this check cannot reveal deviations from the correct result which are multiples of 9, it is so good at catching errors, that it is known as the nine proof.

Similarly, in our physics course, we learned to check the dimensions of formulas retrieved from our unreliable human memory before applying them in solving a problem. For example, when in doubt whether $2\pi r$, or πr^2 is the right formula for computing the area of a circle, we compute their dimensions. Using L for dimension of length, we obtain L^1 for the former and L^2 for the latter. Comparing these results with the dimension of an area, L^2 , we can make the proper choice.

These simple examples suggest the notion of an abstraction function, a total function mapping elements from a concrete domain into elements of an abstract domain and the following definition for (exact) abstraction:

Definition 1.1 Exact Abstraction

Let $f : D_1 \times D_2 \times \dots \times D_n \rightarrow D$ be a function over concrete domains $D_1, D_2, \dots, D_n$ and D ; let $\alpha_i : D_i \rightarrow D_i^\alpha$, ($i = 1, \dots, n$) and $\alpha : D \rightarrow D^\alpha$ be abstraction functions and $f^\alpha : D_1^\alpha \times D_2^\alpha \times \dots \times D_n^\alpha \rightarrow D^\alpha$ a function over the abstract domains. f^α is the exact abstraction of f iff for all $i_1 \in D_1, \dots, i_n \in D_n$ holds: $\alpha(f(i_1, \dots, i_n)) = f^\alpha(\alpha_1(i_1), \dots, \alpha_n(i_n))$. $\square$

Example 1.2 Abstraction of Multiplication

$$\begin{aligned} f = \text{mult} : N \times N &\rightarrow N & \alpha_1 = \alpha_2 = \alpha = \text{mod}_9 : N &\rightarrow N_9 \\ \text{mult}(x, y) &= x * y & \text{mod}_9(x) &= x \text{ modulo } 9 \end{aligned}$$

where the abstract domain N_9 is the set $\{0, 1, \dots, 8\}$; elements are denoted x^α and y^α .

$$\begin{aligned} f^\alpha = \text{mult}^\alpha : N_9 \times N_9 &\rightarrow N_9 \\ \text{mult}^\alpha(x^\alpha, y^\alpha) &= \text{mod}_9(x^\alpha * y^\alpha) \end{aligned}$$

mult^α is an exact abstraction of mult . $\square$

However, this definition is too stringent. A first problem is with partial functions. Consider division and the use of the modulo_9 function as abstraction function for the division. While division by 0 is undefined, division by 9, or multipliers of 9 are defined. But 0 and 9 are indistinguishable after abstraction, so exact abstraction of the division operation is not achievable. A first step towards a solution consists of extending the partial function to be abstracted into a total function. We add an element $\perp$ (undefined) to the concrete domain. The question arises how to abstract $\perp$. By a new element $\perp^\alpha$ in the abstract

domain D^α ? Doing so would make the goal of exact abstraction unachievable as $\perp^\alpha$ cannot be the exact abstraction of the result of a division by 9. The problem has nothing to do with the element $\perp$ but points to a second problem with the notion of exact abstraction.

Consider sign abstraction (abstract domain $\{+, 0^\alpha, -\}$) for numbers. While exact abstraction of multiplication is straightforward, it is not achievable for addition: $++-$ should be $+$ if we consider abstraction of $7 + (-5)$, but 0^α if we consider the abstraction of $7 + (-7)$ and $-$ if we consider the abstraction of $5 + (-7)$. To get out of this impasse, we introduce a new element e.g. $\top$ which is not the abstraction of any number, but a safe approximation of the abstraction of any number, i.e. a safe approximation of $+$, 0^α and $-$. In addition we replace the notion of exact abstraction by the notion of approximate abstraction or safe abstraction.

Similar, in the division example, one has to be satisfied with a result which approximates as well the abstraction of $\perp$ as the result of division by multiples of 9.

Definition 1.3 Safe Approximation

Given domain D equipped with a partial order $\subseteq$; an element d_1 is safe approximation of an element d_2 iff $d_2 \subseteq d_1$. $\square$

Definition 1.4 Safe Abstraction

Let $f : D_1 \times D_2 \times \dots \times D_n \rightarrow D$ be a total function over the concrete domains $D_1, D_2, \dots, D_n$ and D ; let $\alpha_i : D_i \rightarrow D_i^\alpha$, ($i = 1, \dots, n$) and $\alpha : D \rightarrow D^\alpha$ be abstraction functions, $f^\alpha : D_1^\alpha \times D_2^\alpha \times \dots \times D_n^\alpha \rightarrow D^\alpha$ a total function over the abstract domains and $\subseteq$ a partial order over D^α . f^α is a safe abstraction of f iff for all $i_1 \in D_1, \dots, i_n \in D_n$ holds: $\alpha(f(i_1, \dots, i_n)) \subseteq f^\alpha(\alpha_1(i_1), \dots, \alpha_n(i_n))$. $\square$

Example 1.5 Abstraction of Addition

$$\begin{aligned} f = \text{add} : Z \times Z &\rightarrow Z & \alpha_1 = \alpha_2 = \alpha = \text{sign} : Z &\rightarrow Z^\alpha \\ \text{add}(x, y) &= x + y & \text{sign}(x) &= \begin{cases} + & \text{if } x > 0 \\ 0^\alpha & \text{if } x = 0 \\ - & \text{else} \end{cases} \end{aligned}$$

where the abstract domain Z^α is the set $\{+, 0^\alpha, -, \top\}$ and is ordered as follows: $+\subseteq \top$, $0^\alpha \subseteq \top$ and $- \subseteq \top$.

$$f^\alpha = \text{add}^\alpha : Z^\alpha \times Z^\alpha \rightarrow Z^\alpha$$

add^α	$+$	0^α	$-$	$\top$
$+$	$+$	$+$	$\top$	$\top$
0^α	$+$	0^α	$-$	$\top$
$-$	$\top$	$-$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

add^α is a safe abstraction of add . $\square$

¹which we implement by taking, modulo 9, the sum of the digits

Example 1.6

As more elaborate example, to be used later, we consider the safe abstraction of a reduction step in the Martelli-Montanari unification algorithm [MaM82]. The domain of the reduction step is an equation system; this is a set of equations over terms constructed from function symbols and variables. Also $\perp$ is an equation system, it denotes failure. We use the notation $e :: Eqs$ for the multiset² Eqs extended with element e , we use X, Y, Z to denote variables, T and S to denote terms, $var(e)$ to denote the set of variables in expression e and X/T to denote a substitution. This allows us to formulate the reduction step as the following set of reduction rules:

$$\begin{aligned} X &= X :: Eqs \xrightarrow{\text{remove}} Eqs \\ X &= T :: Eqs \xrightarrow{\text{substitute}} \begin{cases} \text{if } X \in var(T) \text{ then } \perp \\ \text{else if } X \in var(Eqs) \text{ then } X = T :: Eqs[X/T] \end{cases} \end{aligned}$$

$$\begin{aligned} f(T_1, \dots, T_n) &= X :: Eqs \xrightarrow{\text{switch}} X = f(T_1, \dots, T_n) :: Eqs \\ f(T_1, \dots, T_n) &= g(S_1, \dots, S_m) :: Eqs \xrightarrow{\text{peel}} \\ &\quad \text{if } f/n = g/m \text{ then } T_1 = S_1 :: \dots :: T_n = S_n :: Eqs \text{ else } \perp \end{aligned}$$

Consider the following abstraction function over terms ($\mathcal{G}$ is a new 0-arity function symbol)

$$\alpha(T) = \begin{cases} \text{if ground}(T) \text{ then } \mathcal{G} \\ \text{else if } T = f(t_1, \dots, t_n) \text{ then } f(\alpha(t_1) \dots \alpha(t_n)) \\ \text{else } \perp \end{cases}$$

This can be extended into an abstraction function over equation systems:

$$\begin{aligned} \alpha(\perp) &= \perp^\alpha \\ \alpha(\emptyset) &= (\mathcal{G} = \mathcal{G}) \\ \alpha(T = S :: Eqs) &= \begin{cases} \text{if ground}(T = S) \text{ then } \alpha(Eqs) \\ \text{else } \alpha(T) = \alpha(S) :: \alpha(Eqs) \end{cases} \end{aligned}$$

The idea is to have a one-to-one mapping between the non-ground equations of the concrete and the abstract system and to abstract the zero or more ground equations of the concrete system by single equations $\mathcal{G} = \mathcal{G}$. In the abstract reduction step below, α is also applied on abstract equation systems. In this trivial extension of α , $\mathcal{G}$ is considered as a ground term.

Now we can formulate the following abstract reduction step (T and S can also denote $\mathcal{G}$):

$$\begin{aligned} X &= X :: Eqs^\alpha \xrightarrow{\text{remove}} Eqs^\alpha \\ X &= T :: Eqs^\alpha \xrightarrow{\text{substitute}} \begin{cases} \text{if } X \in var(T) \text{ then } \perp^\alpha \\ \text{else if } X \in var(Eqs^\alpha) \text{ then } X = T :: \alpha(Eqs^\alpha[X/T]) \end{cases} \end{aligned}$$

²A multiset can have several occurrences of the same element.

$$\begin{aligned} f(T_1, \dots, T_n) &= X :: Eqs^\alpha \xrightarrow{\text{switch}} X = f(T_1, \dots, T_n) :: Eqs^\alpha \\ f(T_1, \dots, T_n) &= g(S_1, \dots, S_m) :: Eqs^\alpha \xrightarrow{\text{peel}} \\ &\quad \begin{cases} \text{if } f/n = g/m = \mathcal{G} \text{ (so } n = m = 0) \text{ then } \mathcal{G} = \mathcal{G} :: Eqs^\alpha \\ \text{else if } f/n = g/m \text{ then } \alpha(T_1 = S_1 :: \dots :: T_n = S_n :: Eqs^\alpha) \\ \text{else if } f/n = \mathcal{G} \text{ then } \alpha(\mathcal{G} = S_1 :: \dots :: \mathcal{G} = S_n :: Eqs^\alpha) \\ \text{else if } g/n = \mathcal{G} \text{ then } \alpha(T_1 = \mathcal{G} :: \dots :: T_n = \mathcal{G} :: Eqs^\alpha) \\ \text{else } \perp^\alpha \end{cases} \end{aligned}$$

Equipped with the partial order $\perp^\alpha \subseteq Eqs^\alpha$ for any abstract equation system Eqs^α , the abstract reduction step is a safe abstraction of the concrete reduction step. However, it is not an exact reduction step. Indeed, consider for example, the concrete system $h(a) = g(a, b) :: Eqs$, assume abstracted as $\mathcal{G} = \mathcal{G} :: Eqs^\alpha$ and a reduction step on the first equation. Concrete reduction yields $\perp$, abstract reduction yields $\mathcal{G} = \mathcal{G} :: Eqs^\alpha$ which is a safe approximation of $\perp$. Notice that one could obtain an exact abstraction by restricting the concrete domain to solvable systems, then $\perp$ and $\perp^\alpha$ are not needed. $\square$

A simpler approach would have been to have no equations $\mathcal{G} = \mathcal{G}$ in the abstract system and to have sets instead of multisets as representation of concrete and abstract systems. However, the close correspondence between a concrete and abstract reduction step would be lost. Indeed, a peel step on a concrete ground equation corresponds to a no-operation on the abstract system. Also, when several non-ground equations are abstracted by the same abstract equation, then a reduction step on that abstract equation corresponds to a number of reduction steps in the concrete system.

1.1.2 Abstracting programs

While manual computations are usually abstracted to catch some of the errors produced by their unreliable implementation in our brains, programs are usually abstracted with a different purpose. One is interested in properties of (intermediate) results which are produced during execution of the program. A well known application is in compiler optimisation. For example, knowing that a predicate in a logic program is always called with its first argument ground allows to specialise the code.

Programs have two features which distinguish them from simple manual computations. Firstly, nontrivial programs have either iterative or recursive constructs, which allow to perform long sequences of primitive operations. Abstracting such a sequence by abstracting each operation in the sequence is problematic. The concrete sequence can be very long, even worse, it can be infinite. As an example consider the well known append predicate (see example 1.11 in section 1.2.1), called with all arguments variables; it produces through backtracking, an infinite number of solutions. Secondly, programs are usually intended to be executed for many different inputs, while a manual computa-

tion is performed for a particular input. It is usually impractical to perform the abstract computation for every allowed input.

A plausible solution for the second problem is to lift f , the operation to be abstracted, to an operation f' between power domains. As will be seen in section 1.1.5, this approach also contains the germ for solving the termination problem.

Example 1.7

Consider the sign abstraction of multiplication. The input of a multiplication consist of a tuple of numbers. The purpose of lifting is to consider a set of tuples. Therefore, we define multiplication as an operation on the domain of tuples of numbers.

$$\begin{aligned} \text{mult} : Z^2 &\rightarrow Z \\ \text{mult}(\langle x, y \rangle) &= x * y \end{aligned}$$

The abstract domain Sign is the set $\{+, 0^a, -\}$. We need two abstraction functions:

$$\begin{aligned} \alpha_1 : Z &\rightarrow \text{Sign} & \alpha_2 : Z^2 &\rightarrow \text{Sign}^2 \\ \alpha_1(n) &= \begin{cases} + & \text{if } n > 0 \\ 0^a & \text{if } n = 0 \\ - & \text{else} \end{cases} & \alpha_2(\langle x, y \rangle) &= \langle \alpha_1(x), \alpha_1(y) \rangle \end{aligned}$$

Lifting mult to power-domains, we obtain ($\mathcal{P}D$ denotes the powerdomain of D):

$$\begin{aligned} \text{mult}' : \mathcal{P}Z^2 &\rightarrow \mathcal{P}Z \\ \text{mult}'(S) &= \{\text{mult}(\langle x, y \rangle) \mid \langle x, y \rangle \in S\} \end{aligned}$$

Lifting the abstraction function we obtain:

$$\begin{aligned} \alpha'_1 : \mathcal{P}Z &\rightarrow \mathcal{P}\text{Sign} & \alpha'_2 : \mathcal{P}Z^2 &\rightarrow \mathcal{P}\text{Sign}^2 \\ \alpha'_1(S) &= \{\alpha_1(n) \mid n \in S\} & \alpha'_2(S) &= \{\alpha_2(\langle x, y \rangle) \mid \langle x, y \rangle \in S\} \end{aligned}$$

The abstraction of mult' is:

$$\begin{aligned} \text{mult}^a : \mathcal{P}\text{Sign}^2 &\rightarrow \mathcal{P}\text{Sign} \\ \text{mult}^a(S) &= \{\text{mult}^a(\langle x, y \rangle) \mid \langle x, y \rangle \in S\} \end{aligned}$$

where $\text{mult}^a : \text{Sign}^2 \rightarrow \text{Sign}$ is defined by the following table:

mult^a	+	0^a	-
+	+	0^a	-
0^a	0^a	0^a	0^a
-	-	0^a	+

□

Power domains are partially ordered by the subset ordering. Even stronger, they are complete lattices. So the lifted abstraction function $\alpha' : \mathcal{P}D \rightarrow \mathcal{P}D^a$ maps elements from a lattice into a lattice (see fig.1.1 for the lattice of $\mathcal{P}\text{Sign}$).

One can make a lot of interesting observations:

Fig. 1.1. Lattice for $\mathcal{P}\text{Sign}$

- every element d of $\mathcal{P}D$ has an abstraction $d^a = \alpha'(d)$ in $\mathcal{P}D^a$.
- elements $d_2^a \supseteq d_1^a = \alpha'(d_1)$ are *safe* approximations of the abstraction of d_1 .
- elements $d_1^a = \alpha'(d_1)$ are *safe* approximations of the abstractions of $d_2 \subseteq d_1$.
- safe approximations d_2^a of an abstraction d_1^a of an element d_1 can also be seen as correct abstractions of d_1 , however, they are not as precise. Every element d_1 has a *best* abstraction $d_1^a = \alpha'(d_1)$.
- one can also consider a function γ from $\mathcal{P}D^a$ to $\mathcal{P}D$, the concretization function, mapping an element d^a to d , the largest set it abstracts. This set is uniquely defined as it is the union of all sets abstracted by d^a .

However, in many cases, it is undesirable to have the full power domain, i.e. the abstract domain could be too big, or could contain too much irrelevant detail. So requesting that the abstract domain is a power domain, or even a subset of a power domain is too rigid. For example, it might be preferable to restrict the domain $\mathcal{P}\text{Sign}$ to the elements $\perp (= \emptyset), +, -, 0^a$. $\top = \{+, -, 0^a\}$.

Also, there is an interest in considering several layers of abstraction where the abstract domain of some layer is the concrete domain of the next layer (an example will be mentioned in section 1.2.1). Consequently, requiring the concrete domain to be a power domain is also too restrictive. However a number of properties are quite essential:

- An order relation between the elements of the concrete domain D . In the sequel, this order relation will be denoted by $\subseteq$ as it is a quite natural case that the concrete domain consists of sets.
- An order relation between the elements of the abstract domain D^a . In the sequel, this order relation will be denoted by $\sqsubseteq$, to distinguish it from the

order relation in the concrete domain, and as the elements of the abstract domain are often not sets.

- The abstraction function α is total, i.e. every element of D has an abstraction in D^α .
- Elements $d^\alpha \sqsupseteq \alpha(d)$ are safe approximations of $\alpha(d)$.
- Elements $\alpha(d_1)$ are safe approximations of $\alpha(d_2)$ with $d_2 \subseteq d_1$.

Others, such as the existence of a *best* abstraction for each element of the concrete domain, and a *best* concretization for each element of the abstract domain, while not essential, are often desirable. An algebraic structure which enforces all the above properties is the Galois connection. Before having a closer look at it, we sum up the basic issues and ideas underlying abstract interpretation of programs.

1.1.3 Basic Issues and Ideas

The theory of abstract interpretation has been developed with the aim of providing a systematic approach to the problem of extracting properties from programs. The basic idea is that the properties of interest are properties of some mathematical function which correctly characterises (certain aspects of) the program. This mathematical function can be a definition of the meaning of the program, independent of a particular execution mechanism, i.e. the declarative semantics such as Tp semantics for logic programs. It can also be a function which characterises substantial aspects of the execution mechanism of the programming language, for example, the SLD semantics of logic programs equipped with the depth-first left-to-right computation rule. One could argue that this function is the definition of the language, or a specification of an implementation. Alternatively, one can see it as an abstraction of an implementation.

As computing this mathematical function for the arguments of interest is in general as infeasible as running the program for the inputs of interest, the goal of abstract interpretation is to approximate this function in a systematic way (in accordance with the intuitions described in the previous sections). Given the programming language and the properties of interest, the idea is to select appropriate abstract domains and to define an abstract function³ which safely approximates the concrete function. Then, given a particular program, the abstract function is computed and this yields correct information regarding the properties of interest. The framework for abstract interpretation gives requirements for the abstract domains, and the abstract function which guarantee that the computation of the abstract function terminates and provides sound information. Choosing abstract domains and abstract functions such that the

³The *abstraction* function is a mapping from the concrete to the abstract domain; the *abstract* function is a function over the abstract domain intended to approximate a function over the concrete domain.

computation is efficient and provides sufficiently precise information belongs to the art of abstract interpretation, and explains why this is such a lively research area. Selecting an abstract domain and defining an abstraction function can be seen as choosing a non-standard domain of data descriptions and defining a non-standard semantics. Computing the abstract function for a particular program can be seen as executing the program over the non-standard domain, using non-standard operations.

1.1.4 Galois connections

As explained above, the concrete domain D is equipped with an order relation $\subseteq$ such that, if $d_1 \subseteq d_2$, then d_2 is a safe approximation of d_1 , i.e. d_2 represents more states than d_1 . Similarly, the abstract domain D^α is equipped with an order relation $\sqsubseteq$ with a similar approximation property: if $d_1^\alpha \sqsubseteq d_2^\alpha$ then d_2^α is a safe approximation of d_1^α , i.e. d_2^α abstracts more states than d_1^α . Both domains are linked to each other by the abstraction and concretization functions. Both are total functions, the former from D to D^α gives the *best* abstraction of elements of D , the latter, from D^α to D gives the *largest* element from D abstracted by a given element from D^α .

Definition 1.8 Galois connection

Given $(D, \subseteq)$, a domain D equipped with a partial order $\subseteq$ and, $(D^\alpha, \sqsubseteq)$, a domain D^α equipped with a partial order $\sqsubseteq$, a Galois connection is a pair of functions α and γ satisfying:

- α is a total function from D to D^α (the abstraction function)
- γ is a total function from D^α to D (the concretization function)
- $\forall d \in D : \forall d^\alpha \in D^\alpha : \alpha(d) \sqsubseteq d^\alpha \text{ iff } d \subseteq \gamma(d^\alpha)$

□

The condition expresses the relationship between both orders and constrains α and γ :

- If d^α approximates the best abstraction of d then the largest set abstracted by d^α is an approximation of d .
- If d is approximated by the largest set abstracted by d^α , then d^α approximates the best abstraction of d .

An alternative definition is often used. The conditions expressed by it can be proven from the conditions expressed by the first definition and vice-versa, so both definitions are equivalent.

Definition 1.9 Galois connection

Given $(D, \subseteq)$, a domain D equipped with a partial order $\subseteq$ and, $(D^\alpha, \sqsubseteq)$, a

domain D^a equipped with a partial order $\subseteq$, a Galois connection is a pair of functions α and γ satisfying:

- α is a total and monotone function from D to D^a
- γ is a total and monotone function from D^a to D
- $\forall d \in D : d \subseteq \gamma(\alpha(d))$
- $\forall d^a \in D^a : \alpha(\gamma(d^a)) \subseteq d^a$

□

This definition requires α and γ to be monotone functions. In addition the third condition requires that the largest concretization of the best abstraction of an element of the concrete domain D is at least as large as the element, i.e. the loss of information due to abstraction is sound, and the fourth condition that the best abstraction of the largest concretization of an element of the abstract domain D^a is at most as large as the element, i.e. concretization introduces no loss of information.

Fig. 1.2. Asymmetry of Galois connections

Notice the asymmetry in the definitions (see fig.1.2 for an illustration). In general, one does not obtain a Galois connection when reversing D and D^a . Typically α^{-1} is not a function, so it cannot be chosen as abstraction function. Taking γ as abstraction function and α as concretization function usually violates the last condition of definition 1.9 as illustrated in fig.1.2. To obtain a Galois connection one also has to invert the order relation.

Many other observations can be made about Galois connections. We list some of them:

- $\gamma(\alpha(\gamma(d^a))) = \gamma(d^a)$ and $\alpha(\gamma(\alpha(d))) = \alpha(d)$. The former indicates that no information is lost by abstracting the concretization of an abstract element, the latter that all information lost by abstraction is lost at once.
- The choice of α determines the choice of γ and vice-versa. So the definition of the second function can be left implicit.
- α preserves arbitrary many least upper bounds, i.e.

$$\alpha(\text{lub}(d_1, \dots, d_n)) = \text{lub}(\alpha(d_1), \dots, \alpha(d_n))$$

and γ preserves arbitrary many greatest lower bounds, i.e.

$$\gamma(\text{glb}(d_1^a, \dots, d_n^a)) = \text{glb}(\gamma(d_1^a), \dots, \gamma(d_n^a))$$

- Elements d^a in D^a such that $\alpha(\gamma(d^a)) \subseteq d^a$ can be considered as superfluous elements of the abstract domain, as they represent the same element in the concrete domain as a strictly smaller element. In the absence of such superfluous elements, i.e. if for all d^a holds that $\alpha(\gamma(d^a)) = d^a$, we have a Galois surjection, sometimes also called a Galois insertion (γ is one-to-one and α is onto).
- In a Galois surjection, the order of the concrete domain is induced in the abstract domain. For example, if the concrete domain is a complete lattice (e.g. a power domain ordered by the subset relation then the abstract domain is also a complete lattice).

It is not impossible to compute safe approximations by means of an algebraic structure that is weaker than a Galois connection, but it will not have all the elegant properties. However, in some cases, it may be more precise, i.e. computing over domains D and D^a which are not linked by a Galois connection can be more precise than computing over subdomains $D_1 \subseteq D$ and $D_1^a \subseteq D^a$ which are linked by a Galois connection.

1.1.5 Approximating the semantics of programming languages

The standard approach to perform abstract interpretation of programs is to define the behaviour of the program as the least fixpoint of a monotone function over some domain of program properties equipped with a partial order. This function must capture those parts of the behaviour which are of interest and either can be a definition of the semantics of the program or has to be proven correct with regard to the semantics of the program. Developing an abstract interpretation consists of choosing a second non-standard domain, defining a Galois connection between both domains, and a monotone function over the non-standard domain approximating the original function. In this way, the steps performed in computing the fixpoint of the original function are approximated by the function computing over the non-standard domain. Computing (safe

approximations of) the fixpoint is easier when the function is also continuous. We will make this assumption in the remainder of the text.

Example 1.10 T_P semantics

Let P be a definite logic program, B_P its Herbrand base and $\text{ground}(P)$ the set of all ground instances of clauses in P . Then T_P is a function from the powerset of the Herbrand base to the powerset of the Herbrand base and is defined as: $T_P(S) = \{A \mid A \leftarrow B_1, \dots, B_n \in \text{ground}(P) \text{ and } \forall i \in [1, n] : B_i \in S\}$. The T_P semantics gives the least Herbrand model of the program and is the least fixpoint of the T_P function: $\text{lfp } T_P = \bigcup_{n \geq 0} T_P^n(\emptyset)$. $\square$

Having a concrete domain D , an abstract domain D^α , a Galois connection between them and a concrete semantics defined as the least fixpoint of a function F over D , the purpose of abstract interpretation is to compute $\alpha(\text{lfp } F)$, or short of that, a safe approximation of it.

The simplest and most accurate way is to compute $\text{lfp } F$ and to abstract it. Unfortunately, in most cases, computing the least fixpoint requires an unbounded number of steps. So, a different approach has to be followed. Assuming the least fixpoint of F is given by $\text{lub}_{n \geq 0}(F^n(\perp))$, with $\perp$ the infimum of D , we will approximate the least fixpoint by the least fixpoint of a monotone function F^α over the abstract domain.

The most precise approach is to have an infimum $\perp^\alpha$ in the abstract domain with $\perp^\alpha = \alpha(\perp)$ and to define F^α as $F^\alpha(d^\alpha) = \alpha(F(\gamma(d^\alpha)))$. As F^α is monotonic, it has a least fixpoint. Unfortunately, the least fixpoint of F^α is not necessarily the abstraction of the least fixpoint of F . Indeed, it is possible that $\gamma(\alpha(\text{lfp } F)) \supset \text{lfp } F$ and thus that $\alpha(F(\gamma(\alpha(\text{lfp } F)))) = F^\alpha(\alpha(\text{lfp } F)) \supset \alpha(\text{lfp } F)$, i.e. $\alpha(\text{lfp } F)$, is not a fixpoint of F^α . However the least fixpoint of F^α is the abstraction of some fixpoint of F ; in the worst case, it is the abstraction of the greatest fixpoint (see fig.1.3 for an illustration).

This approach can still be impractical. In the common case where the concrete domain is a power domain, $\gamma(d^\alpha)$ is a set which can be infinite. If $F(\gamma(d^\alpha))$ is defined by applying some function F' on each element of $\gamma(d^\alpha)$, then it is impossible to compute $F(\gamma(d^\alpha))$. In such a situation, it is necessary to define F^α directly over D^α . The ideal is to have: $\forall d^\alpha : F^\alpha(d^\alpha) = \alpha(F(\gamma(d^\alpha)))$, however, a safe approximation is sufficient, i.e. $\forall d^\alpha : F^\alpha(d^\alpha) \supseteq \alpha(F(\gamma(d^\alpha)))$.

Having replaced the problem of approximating the least fixpoint of the concrete semantic function F over the concrete domain D by the problem of finding the least fixpoint of the abstract semantic function F^α over the abstract domain D^α has not solved by itself the termination problem. However, termination is assured if the abstract domain has no infinite ascending chains. i.e. if D^α has no infinite sequence of elements $d_0^\alpha \subset d_1^\alpha \subset d_2^\alpha \subset \dots$. Indeed, the number of steps in computing a fixpoint is bounded by the length of the longest chain. In the presence of infinite ascending chains, termination can still be enforced by so-called widening. The idea is to replace $F^\alpha(d^\alpha)$ by $F^\alpha(d^\alpha) \nabla d^\alpha \supseteq F^\alpha(d^\alpha)$ where ∇ is a so-called widening operator which assures that in a finite number

Fig. 1.3. F^α defined as $\alpha \circ F \circ \gamma$

of steps a post fixpoint is reached, i.e. a point d^α for which $F^\alpha(d^\alpha) \subseteq d^\alpha$ and which is a safe approximation of the fixpoint. (The simplest way is to define ∇ as: $F^\alpha(d^\alpha) \nabla d^\alpha = \text{if number of iteration below threshold then } F^\alpha(d^\alpha) \text{ else } \top$ with $\top$ the supremum of D^α). Widening can also be used in case of finite ascending chains to further reduce the number of steps in the fixpoint computation.

Applying widening can overshoot the fixpoint of F^α , it is possible to reach a point d^α where $F^\alpha(d^\alpha) \subset d^\alpha$. A better approximation can be obtained by applying so-called narrowing on the monotone F^α . Roughly speaking, the idea is to apply F^α a finite number of times to obtain a better approximation of the fixpoint of F^α .

Widening and narrowing enforce termination by ensuring that only a finite subspace of the infinite abstract domain is explored. This is in general better than starting with a finite abstract domain as each program being analysed can explore a different finite part of the infinite abstract domain and the union of all finite parts for all possible programs can easily be infinite [CoC92c].

Patrick and Radhia Cousot have pioneered the formal development of abstract interpretation. The recent papers [CoC92a] and [CoC92b] summarise the results and contain an extensive list of references to the original papers (and were the main source of inspiration for the material in this section).

1.2 Abstract interpretation for logic programs

1.2.1 Abstracting T_p semantics

The least Herbrand model of a program P is given by the least fixpoint of the T_p function as defined in example 1.10. In this section we illustrate the ideas by developing an abstraction of the T_p semantics. The purpose of the abstraction is to derive so-called size relations. Size relations express relationships between arguments of atoms in the least Herbrand model. For example, using the length of a list as its size, and considering append (see example 1.11 below) atoms in the least Herbrand model, we have that $size(arg_1) + size(arg_2) = size(arg_3)$.

Size relations are of interest in proving termination of programs [VDS91, VDS92], they can also be interesting for compiler optimisations, for example, a list of known length can be implemented as an array. In constraint logic programming, they can be used to make implicit constraints explicit. However, the main point here is that they are an elegant abstraction of T_p semantics.

In what follows, we discuss the abstract domain for a single predicate. It is straightforward to generalise the introduced notions to a vector whose elements are the domain elements of individual predicates.

First we define a function $size$ which maps a term in the Herbrand Universe into a natural number. Assuming the size of interest is $listlength$ (another possibility is the number of function symbols), we define:

$$\begin{aligned} size : \text{Herbrand Universe} &\rightarrow N \\ size(t) &= \text{if } (t = [x|y]) \text{ then } 1 + size(y) \text{ else } 0 \end{aligned}$$

The function can be extended in a function mapping atoms from the Herbrand Base of a predicate p/m into m -tuples:

$$\begin{aligned} size : \text{Herbrand Base of } p/m &\rightarrow N^m \\ size(p(t_1, \dots, t_m)) &= \langle size(t_1), \dots, size(t_m) \rangle \end{aligned}$$

In this way, a Herbrand interpretation of predicate p/m is mapped into a set of m -dimensional points. These sets of m -dimensional points make up a complete lattice ordered by the subset relation. As abstract domain we choose systems of linear equations. Such a system consists of sets of equations of the form $a_1x_1 + \dots + a_mx_m = b$. They define *affine subspaces*, also called *linear varieties* of the m -dimensional space. Given a system E^p for predicate p/m , its concretization is the set of ground atoms whose size tuple is a solution of E^p . Formally:

$$\gamma(E^p) = \{p(t_1, \dots, t_m) \mid \langle size(t_1), \dots, size(t_m) \rangle \text{ is a solution for the equations in } E^p\}$$

This concretization function induces a preorder relation (reflexive and transitive) over the systems of equations E^p :

$$E_1^p \subseteq E_2^p \text{ iff } \gamma(E_1^p) \subseteq \gamma(E_2^p)$$

Defining equivalence as:

$$E_1^p \approx E_2^p \text{ iff } \gamma(E_1^p) \subseteq \gamma(E_2^p) \text{ and } \gamma(E_2^p) \subseteq \gamma(E_1^p)$$

we obtain a partial order between equivalence classes. For convenience, we introduce $\perp$ with $\gamma(\perp) = \emptyset$, i.e. an unsolvable system and $\top$ with $\gamma(\top) = \{p(t_1, \dots, t_m)\}$, i.e. the system representing the complete Herbrand Base of the predicate. This abstract domain is infinite, however, the length of ascending chains is bounded by the arity of the predicate. A fragment of the abstract domain for a 3-argument predicate is shown in fig.1.4.

Fig. 1.4. Part of the Infinite Abstract Domain

We have a Galois connection. The order relations together with the concretization function, induce the abstraction function. Given a set of ground atoms $p(t_1, \dots, t_m)$, the abstraction is a representative from the least equivalence class E^p having the points $\langle size(t_1), \dots, size(t_m) \rangle$ as a solution. A representative of the glb of two systems of equations is given by the system

containing the equations of both systems. Computing the lub of two systems is less trivial. The lub of two different points is the line connecting the points, the lub of a line and a point outside the line is a plane, An algorithm for computing a representative of the lub class is given in [Karr76].

Notice that we could have formulated our construction as a two layered abstraction. At the concrete level we have sets of atoms from the Herbrand Base of p/m , at the first level of abstraction we have sets of points in a m -dimensional space and at the second level, we have the systems of equations. All levels can be connected to each other by Galois connections.

To perform a manual analysis as in the next example, we define T_p^a as $\alpha \circ T_p \circ \gamma$.

Example 1.11 Size Relation for Append

The program for `append/3` is:

```
append([], X, X) ←
append([X|U], V, [X|W]) ← append(U, V, W)
```

As initialisation, we have:

$$T_p \uparrow 0 = \emptyset$$

$$T_p^a \uparrow 0 = \perp^a$$

while $\alpha(\emptyset) = \perp^a$ and $\gamma(\perp^a) = \emptyset$.

$$T_p^a \uparrow 1 = \alpha(T_p(\gamma(\perp^a)))$$

$$= \alpha(T_p(\emptyset))$$

$$= \alpha(\{\text{append}([], t, t)\})$$

The function `size` maps these atoms into the points $\langle 0, 0, 0 \rangle$, $\langle 0, 1, 1 \rangle$, $\langle 0, 2, 2 \rangle$, ..., $\langle 0, n, n \rangle$, The minimal system of equations having these points as solutions describes a line going through the points, so

$$T_p^a \uparrow 1 = \{x_1 = 0, x_2 = x_3\}$$

$$T_p^a \uparrow 2 = \alpha(T_p(\gamma(\{x_1 = 0, x_2 = x_3\})))$$

$$= \alpha(T_p(\{\text{append}(t_1, t_2, t_3) \mid \text{size}(t_1) = 0, \text{size}(t_2) = \text{size}(t_3)\}))$$

Notice that $\gamma(\{x_1 = 0, x_2 = x_3\}) \supseteq \{\text{append}([], t, t)\}$.

$$T_p^a \uparrow 2 = \alpha(\{\text{append}([], t, t)\} \cup$$

$$\{\text{append}([t_X|t_U], t_V, [t_X|t_W]) \mid \text{size}(t_U) = 0, \text{size}(t_V) = \text{size}(t_W)\})$$

Because we have a Galois connection, we can perform the lub in the abstract domain ($\alpha(a \cup b) = \text{lub}(\alpha(a), \alpha(b))$):

$$T_p^a \uparrow 2 = \text{lub}(\alpha(\{\text{append}([], t, t)\}),$$

$$\alpha(\{\text{append}([t_X|t_U], t_V, [t_X|t_W]) \mid \text{size}(t_U) = 0, \text{size}(t_V) = \text{size}(t_W)\}))$$

The atoms contributed by the second clause are mapped by the function `size` into the points $\langle 1, 0, 1 \rangle$, $\langle 1, 1, 2 \rangle$, ..., $\langle 1, n, n+1 \rangle$, The abstraction of this set of points is the equation of the line going through them, so

$$T_p^a \uparrow 2 = \text{lub}(\{x_1 = 0, x_2 = x_3\}, \{x_1 = 1, x_2 + 1 = x_3\})$$

The upper bound of these two parallel lines is a plane:

$$T_p^a \uparrow 2 = \{x_1 + x_2 = x_3\}$$

$$T_p^a \uparrow 3 = \alpha(T_p(\gamma(\{x_1 + x_2 = x_3\})))$$

$$= \alpha(T_p(\{\text{append}(t_1, t_2, t_3) \mid \text{size}(t_1) + \text{size}(t_2) = \text{size}(t_3)\}))$$

$$= \alpha(\{\text{append}([], t, t)\} \cup$$

$$\{\text{append}([t_X|t_U], t_V, [t_X|t_W]) \mid \text{size}(t_U) + \text{size}(t_V) = \text{size}(t_W)\})$$

$$= \text{lub}(\alpha(\{\text{append}([], t, t)\}),$$

$$\alpha(\{\text{append}([t_X|t_U], t_V, [t_X|t_W]) \mid \text{size}(t_U) + \text{size}(t_V) = \text{size}(t_W)\}))$$

Observe that atoms contributed by the second clause are mapped into points of the same plane $x_1 + x_2 = x_3$, so

$$T_p^a \uparrow 3 = \text{lub}(\{x_1 = 0, x_2 = x_3\}, \{x_1 + x_2 = x_3\})$$

$$= \{x_1 + x_2 = x_3\}$$

So a fixpoint is reached. Notice that computing the fixpoint of T_p requires an infinite number of steps. $\square$

The analysis can be automated without much difficulty. A key step is to make unification explicit by calls to the equality predicate `=/2` which is interpreted as syntactical identity. Then T_p^a can be defined directly on the abstract domain, as a function mapping a vector of systems of equations — one for each predicate — into a vector of systems of equations. The abstraction of calls to predicates different from equality are obtained by renaming the system in the given vector. Full details can be found in [VDS91, VDS92] (although in a top-down setting). We sketch the idea by some examples.

The first clause of `append/3` is rewritten as:

$$\text{append}(X_1, X_2, X_3) \leftarrow X_1 = [], X_2 = X_3$$

The calls to `=/2` are abstracted respectively as $\{x_1 = 0\}$ and $\{x_2 = x_3\}$. The glb of both systems is given by their conjunction, i.e. $\{x_1 = 0, x_2 = x_3\}$ which is the abstraction of the atoms contributed by the clause.

The second clause is rewritten as:

$$\text{append}(X_1, X_2, X_3) \leftarrow X_1 = [X|U], X_3 = [X|W], \text{append}(U, X_2, W)$$

At the beginning of the second iteration, the current abstraction of `append/3` in the vector is $\{x_1 = 0, x_2 = x_3\}$. The recursive call is abstracted by a renaming of this, i.e. $\{u = 0, x_2 = w\}$. The calls to `=/2` are abstracted respectively by $\{x_1 = 1 + u\}$ and $\{x_3 = 1 + w\}$. The glb of these systems is

$$\{u = 0, x_2 = w, x_1 = 1 + u, x_3 = 1 + w\}$$

which is equivalent to

$$\{u = 0, x_2 = w, x_1 = 1, x_3 = 1 + x_2\},$$

projection upon x_1, x_2, x_3 gives $\{x_1 = 1, x_3 = 1 + x_2\}$ which is the abstraction of the atoms contributed by the second clause during the second iteration.

Observe that one could take an alternative view where the abstract domain consists of sets of affine subspaces and where widening is used to limit the cardinality of the computed abstract elements. By limiting cardinality to 1, we obtain exactly the computations described above. This illustrates that computing in a richer domain need not be more expensive when widening is used.

Finally, it is interesting to remark that one could use a richer abstract domain containing also inequalities $a_1x_1 + \dots + a_mx_m \geq b$. Such systems define solution spaces which are convex hulls. The abstract domain becomes of infinite height and widening is needed to ensure termination. Such an abstract domain has been used in [CoH78] for deriving properties of imperative programs.

1.2.2 OLD Semantics

T_P semantics captures only information about the least Herbrand model of the program, about the result of a computation. To obtain information about intermediate states in the computation, i.e. information about the calls in a Prolog program, a different semantics is needed. Here we study OLD semantics. OLD, introduced by Sato [TaS86], is SLD where the computation rule is restricted to select always the leftmost literal (as Prolog does). Concepts such as OLD derivation, OLD refutation and OLD tree are obvious adaptations from the similar concepts for SLD [Llo87].

To define the semantics as the fixpoint of an operator mapping partial OLD trees into partial OLD trees, we introduce the notion of a (partial) OLD tree.

Definition 1.12 Partial OLD tree

Let P be a definite program and G a definite goal. A partial OLD tree for $P \cup \{G\}$ is a tree satisfying the following:

- Each node of the tree is a (possibly empty) definite goal.
- The root node (if any) is G .
- Let $\leftarrow A_1, \dots, A_m$ ($m \geq 1$) be a node in the tree that is not a leaf. Then for each program clause $A \leftarrow B_1, \dots, B_k$ such that A_1 and A unify with $mgu \theta$, the node has a child $\leftarrow (B_1, \dots, B_k, A_2, \dots, A_m)\theta$ (these children are the result of expanding the node).
- Nodes which are empty goals are leaves.

A leaf node $\leftarrow A_1, \dots, A_m$ such that there exists a program clause $A \leftarrow B_1, \dots, B_k$ such that A_1 and A unify is called an unexpanded leaf. $\square$

Now we can define a monotone operator $F_{P,SG}$ for a program P and a set of goals SG which maps a set of partial OLD trees into a set of partial OLD trees as follows:

$$F_{P,SG}(ST) = \{t \mid t \text{ has a single node consisting of a goal from } SG\} \cup \{t \mid t' \in ST \text{ and } t \text{ is derived from } t' \text{ by expanding all unexpanded leaves}\}$$

The OLD semantics consists of the least fixpoint of the operator $F_{P,SG}$. It computes information about all goal statements which are processed while executing a query from the set SG , in particular it shows all instances of called predicates.

A simple minded abstraction of the domain of OLD trees would keep the tree structure and abstract the goals. It is obvious that this suffers from termination problems as the depth of the trees can be unbounded. Therefore, it is more convenient to first modify the OLD execution mechanism before abstracting it.

1.2.2.1 OLDT Resolution OLDT resolution stands for OLD resolution with tabulation and has been introduced by Tamaki and Sato [TaS86] as a run-time engine to improve standard Prolog interpreters. A little later it was also introduced by Vieille as the SLD-AL method for execution of deductive database queries [Vie89]. Recently OLDT resolution is widely used as an abstract interpretation engine [KaK93, CoF92].

OLD resolution is SLD resolution with a restricted computation rule: only the first atom of the current goal can be selected. This restriction upon the computation rule is a crucial point for the correctness of tabulation (in general, tabulation is not applicable to SLD resolution – certain restrictions upon the computation rule are necessary).

The idea of tabulation is very simple. All call patterns of the selected atoms of the goals met so far in the derivation are collected in a table. Also (different) answers to the calls are stored in the table. When a call has to be performed and its pattern is already in the table, then instead of using the clauses of the program, the answers stored in the table are used as facts defining the called predicate. So entries of the table consist of two parts: a call pattern and a list of corresponding answers which are instances of the call pattern.

Example 1.13

Given the definite logic program

$c_1 : \text{path}(X, Y) \leftarrow p(X, Y)$
 $c_2 : \text{path}(X, Y) \leftarrow \text{path}(X, Z), p(Z, Y)$
 $c_3 : p(a, b).$
 $c_4 : p(b, c).$
 $c_5 : p(c, a).$

and the initial goal $\leftarrow \text{path}(a, Y)$. The corresponding OLD T tree is shown in fig.1.5. The argument i in $\text{lookup}(i)$ refers to the number of the first node with the same call pattern. The indices (i, j) in A_{ij} refer to the position in the table of answers below.

Fig. 1.5. OLD T Tree

The table entries created during execution are:

call	answers		
	1	2	3
1 $\text{path}(a, U)$	$\text{path}(a, b)$	$\text{path}(a, c)$	$\text{path}(a, a)$
2 $p(a, U)$	$p(a, b)$		
5 $p(b, U)$	$p(b, c)$		
7 $p(c, U)$	$p(c, a)$		

□

The calls resolved with the facts in the table are called *lookup* calls (e.g. nodes 4 and 9 in fig. 1.5). The most important property of OLD T resolution is

that it is sound and complete. Moreover, given an OLD tree, the corresponding OLD T tree has exactly the same call/answer patterns. The latter is very important when applying OLD T resolution for abstract interpretation. It is intuitively obvious that an OLD T tree can be expanded into a full OLD tree by replacing *lookup* extensions by the corresponding properly renamed subrefutations. In general, this process can be infinite as the replacement can introduce again *lookup* nodes. This is the case for the above example 1.13.

An OLD T tree can also be infinite. However, it can rather easily be abstracted in a finite structure. A generic abstraction scheme for OLD T resolution should specify at least the following components:

- **Goal abstraction:** whenever a new goal is obtained, it is abstracted. The abstraction must be such that the concretization includes the original goal. The next OLD T step uses the abstracted goal. In particular, the abstracted call either becomes a new entry in the table or is used to locate the appropriate entry to be used in a *lookup* step. Another requirement on the abstraction is that only a finite number of table entries are created.
- **Answer abstraction:** Whenever an answer has been obtained, it is abstracted before storing it in the table. Again, the abstraction must be such that the concretization includes the original answer and that only a finite number of answers are possible.

Having a Galois connection linking sets of concrete goals/answers with abstract goals/answers, we can obtain *AOLD T*, the abstract counterpart of OLD T.

Example 1.14

Using the abstract equation systems of example 1.6, we can represent abstract goals and answers by a pair consisting of a concrete goal/answer and an abstract equation system. For example the pair $\text{append}(X, Y, Z), \{X = G, Y = G\}$ represents $\text{append}(X, Y, Z)\{X \leftarrow t_1, Y \leftarrow t_2\}$ where t_1, t_2 are ground terms. (To obtain a Galois connection between the domains of concrete and abstract terms, we should introduce an abstract term *Any* representing the whole Herbrand universe; a widening operator should introduce *Any* when terms are too deeply nested and abstract unification, as defined in example 1.6, has to be extended with rules coping with *Any*. However, in the examples we consider here, widening and *Any* are not needed).

Starting from an abstract goal $\leftarrow \text{append}(X, Y, Z), \{X = G, Y = G\}$, one can work out an abstract OLD T tree and one can observe that it is finite (see fig.1.6). The boxes labelling the branches are either the clauses used in node expansion or answers used in *lookup*. The table has only a single entry: $\text{append}(X, Y, Z)\{X = G, Y = G\} \mid \{X = G, Y = G, Z = G\}$ (The second abstract equation system specifies how the calls are instantiated by the answers).

□

Fig. 1.6. Abstract OLDT Tree

An abstract *OLDT* tree can be expanded into an abstract *OLD* tree, in the same way as an *OLDT* tree can be expanded into an *OLD* tree. Doing this for the tree of the *append* program will result in an infinite tree which abstracts any *OLD* tree obtained from a query $\leftarrow \text{append}(X, Y, Z), \{X = t_1, Y = t_2\}$ where t_1, t_2 are ground terms.

Abstract *OLDT* has the inconvenience that it somehow computes too detailed information. Consider a goal $\leftarrow A_1, A_2$ somewhere in an *AOLDT* tree such that it is the body of some program clause. Due to the non-determinism of logic programs, the subtree rooted at this node will usually have different branches. Once A_1 is refuted, one finds the goal $\leftarrow A_2$ on different branches. Usually, these goals will have been instantiated differently. *AOLDT* will perform an analysis for each of them. However, in many applications, i.e. compilation, one wants to have a single analysis for A_2 , starting from a call pattern which is general enough to describe the calls to A_2 on the different branches. It is more economical to do a single analysis for the more general pattern than to perform an analysis for each pattern and then to combine the results. While one could

modify *AOLDT* to handle the situation, there is another approach which avoids this problem in a more natural way.

Fig. 1.7. AND - OR graph

1.2.2.2 Local SLD with tabulation: *LSLDT* The basic step of *SLD* is to select a literal in a goal and a clause in the program, to replace the literal by the clause body and to apply the *mgu* of the literal and the clause head. Here we define a variant of *SLD* which we call local *SLD* (*LSLD*). Given a goal $\leftarrow A_1, \dots, A_n$, a literal A_i is selected and for each clause $H \leftarrow B_1, \dots, B_m$ such that A_i and H have *mgu* θ , a subrefutation $\leftarrow (B_1, \dots, B_m)\theta$ is initiated. Eventually, this subderivation computes an answer τ (through backtracking several ones). By soundness of *SLD*, $H\tau$ is a logical consequence of the program,

i.e. it can be considered as a lemma. Now, one returns to the main computation with the goal $\leftarrow A_1, \dots, A_n$. The call A_i is selected and the lemma $H\tau$ is used to resolve it.

Notice that we restrict the computation rule in a way very similar to *OLD*. Indeed we divide the computation into many subcomputations one for each call, and return to the caller's computation only when the subcomputation has completed by computing an answer. This corresponds to *SLD* where the computation rule always selects one of the most recently introduced literals. In the sequel we will for simplicity assume always the leftmost literal is selected, as in *OLD*.

We call the resulting mechanism *LSLD* because the computation is splitted into many local parts. A subderivation corresponds closely to the viewpoint of a compiler writer generating code for a predicate definition: different clauses defining the predicate are tried, and the clause bodies are executed by initiating calls to the different body literals. Each of them creates another subderivation. So it is no surprise that *LSLD* is a good starting point for an analysis aimed at providing compiler writers with useful information.

To avoid the problem of an unbounded number of different subderivations, *LSLD* can be enhanced with tabulation, in the same way as *OLD* was. The call patterns in the table are the goals which are at the roots of the subderivations. Similar as for *OLDT*, each entry has a list of answers, the lemmas. When a new subderivation is created, the table entries are checked. If the created subderivation is in the table, then it is cancelled and the stored lemmas are used to resolve the call; if not, a new table entry is created and the subderivation is solved⁴.

Now abstraction of *LSLDT* is very similar to abstraction of *OLDT*, i.e. goal statements and answers are abstracted. As the table can have different answers for a particular call, at first glance we still suffer from the same inconvenience as *AOLDT*, that we have different continuations for the same residual part of clause bodies (the *LSLD* tree branches, one branch for each answer). However, this branching can be avoided in a very natural way. The abstract domain of answers for a given literal form a partial order, and each set of answers has an upper bound. So, in executing a goal statement $\leftarrow A_1, A_2, \dots$, the upper bound of all answers to A_1 is used as a single lemma, this assures there is a single continuation $\leftarrow A_2, \dots$.

However, there is still a subtle problem when performing an analysis based on this scheme: it is possible one first uses a lemma for solving the literal A_1 in $\leftarrow A_1, A_2, \dots$, derives a new goal $\leftarrow A_2, \dots$, and next, creates a new subcomputation for $\leftarrow A_2$. At a later point, another solution for the goal $\leftarrow A_1$ can be obtained, which is not smaller than the lemma which has been used. At this point a new least upper bound for solutions to A_1 is obtained and the deriva-

tion starting from $\leftarrow A_1, A_2$ has to be redone. Notice that this implies that the original entry in the table for A_2 becomes useless, as well as the subderivation it was initiating (and may have created other table entries).

A high level data structure which integrates all subderivations into a single structure and, at a high level, addresses the problem of avoiding useless table entries and useless subcomputations is the *AND-OR* graph used in the framework of Bruynooghe [Bru91]. The unexpanded calls at the leaves of the graph are nothing else than calls resolved by table look-up. Somewhere else in the tree, there is an equivalent call which has been expanded by entering all defining clauses and whose answers have been collected into a single upper bound. The *AND-OR* graph structure is instrumental in (1) allowing different specialised versions of the same clause and indicating which version has to be called at which point, (2) avoiding useless table entries and useless subderivations, (3) organising the computation of safe approximations (fixpoints) in an optimal way, (4) providing a high level interface with the compiler.

Example 1.15

c_1 : $\text{delete}(X, L, R) \leftarrow L = [X|R]$.
 c_2 : $\text{delete}(X, L, R) \leftarrow L = [Y|L'], R = [Y|R'], \text{delete}(X, L', R')$.
 c_3 : $\text{perm}(L, PL) \leftarrow L = [], PL = []$.
 c_4 : $\text{perm}(L, PL) \leftarrow PL = [X|P], \text{delete}(X, L, R), \text{perm}(R, P)$.

Query : $\leftarrow \text{perm}(A, B), \{A = G\}$

creation of subrefutations (a) and (b)

$\leftarrow \Diamond \{A = G, B = G\}$ (using L1 below)

(a) subderivation body c_3

$\leftarrow L = [], PL = [] \{L = G\}$
 $\leftarrow PL = [] \{L = G\}$
 $\leftarrow \Diamond \{L = G, PL = G\}$

Lemma : $\text{perm}(L, PL) \{L = G, PL = G\}$ (L1)

(b) subderivation body c_4

$\leftarrow PL = [X|P], \text{delete}(X, L, R), \text{perm}(R, P) \{L = G\}$
 $\leftarrow \text{delete}(X, L, R), \text{perm}(R, P) \{L = G, PL = [X|P]\}$

creation of subrefutations (c) and (d)

$\leftarrow \text{perm}(R, P) \{L = G, PL = [G|P], X = G, R = G\}$ (using L2)
 $\leftarrow \Diamond \{L = G, PL = G, X = G, R = G, P = G\}$ (using L1)

Same lemma as L1

(c) subderivation body c_1

⁴In practice it is more convenient to table the literal which triggered the creation of a subderivation; the information showing which clause contributed which lemma is lost, but this information is not needed. However, the overhead of creating subderivations is avoided in a case of a lookup.

$$\begin{aligned} &\leftarrow L = [X|R] \quad \{L = G\} \\ &\leftarrow \Diamond \{L = G, X = G, R = G\} \end{aligned}$$

Lemma: $\text{delete}(X, L, R) \{X = G, L = G, R = G\}$ (L2)

(d) subderivation body c_2

$$\begin{aligned} &\leftarrow L = [Y|L'], R = [Y|R'], \text{delete}(X, L', R') \{L = G\} \\ &\leftarrow R = [Y|R'], \text{delete}(X, L', R') \{L = G, Y = G, L' = G\} \\ &\leftarrow \text{delete}(X, L', R') \{L = G, Y = G, L' = G, R = [G|R']\} \\ &\leftarrow \Diamond \{L = G, Y = G, L' = G, R = G, X = G, R' = G\} \text{ (using L2)} \end{aligned}$$

Same lemma as L2

The *AND-OR* graph summarising the computation is shown in fig.1.7. The structure is called a graph because the *lookup* nodes can be considered as arcs pointing to the nodes with the same call pattern. Between $||$ one can observe the abstract states as computed in the subderivations. The nodes $p(R, P)$ and $d(X', L'', R'')$ are leaves as they are solved by table *lookup* (use of lemmas). $\square$

The *AND-OR* graphs represent, via the concretization function, a set of *LSLDT* derivations. These derivations in turn represent *OLD* derivations. The *AND-OR* graph is the fixpoint of a function mapping partial *AND-OR* graphs into partial *AND-OR* graphs. It can be shown this fixpoint represents a fixpoint of the function $F_{P,SG}$ introduced in section 1.2.2, which maps sets of partial *OLD* trees into sets of partial *OLD* trees [Bru91]. The *AND-OR* graph is the conceptual basis for several implemented frameworks such as PLAI [MuH92] and GAIA [CVH91, CVH92].

In abstracting *OLDT* and abstracting *LSLDT*, we can distinguish a couple of crucial operations. To bring them more to the foreground and in view of the extension to constraint logic programming, it is useful to separate goals and literals in two parts: a skeleton and a substitution. A skeleton is of the form $p(X_1, \dots, X_n)$ with the X_i different variables. The substitution can be written as a solved form of a system of equations [LMM88]. We use $eq(\theta)$ to denote the solved form corresponding to the substitution θ .

A first crucial operation is unification. In both *OLDT* and *LSLDT*, one has to unify a call $\langle A, eq(\theta) \rangle$ with a clause head H . It means one has to bring into solved form the system $\{eq(\theta), A = H\}$. In both *OLDT* and *LSLDT*, one has also lemma application (*lookup*), i.e. a call $\langle A, eq(\theta) \rangle$ and a lemma $\langle A', eq(\tau) \rangle$ have to be unified. So, abstracting unification in the most general setting, one has a literal A with associated abstract system of equations E_A , and a literal B with associated system E_B (no sharing of variables), and the purpose is to compute a new system E which approximates the system $\{E_A, A = B, E_B\}$. More formally, one can formulate a correctness condition as follows: $\forall eq(\theta) \in \gamma(E_A), \forall eq(\sigma) \in \gamma(E_B)$: if $\text{vars}(A\theta) \cap \text{vars}(B\sigma) = \emptyset$ and $\{eq(\theta), A = B, eq(\sigma)\}$ has $eq(\tau)$ as solved form then $eq(\tau) \in \gamma(E)$.

Another operation is concerned with restricting the domain of substitutions. While we have pairs $\langle A, eq(\theta) \rangle$, what is really important is the instance of

A , i.e. $A\theta$. So one can drop those components of θ that do not affect $A\theta$, in other words, the substitution can be restricted to the variables of A . While this operation passes almost unnoticed at the concrete level, it is not unimportant when considering the table entries. When separating the table entries also in pairs skeleton-solved form, it is crucial that the solved form is restricted to the variables of the skeleton, otherwise, matches of new calls with existing table entries will go unnoticed. The operation becomes more prominent at the abstract level, not only to avoid mismatches when checking new calls, but also to restrict the size of the abstract equation systems, i.e. the number of elements in the abstract domain. Notice that this restriction can also be seen as existential quantification, indeed in a component $X = t$ in a solved form one can hide the X by existential quantification $\exists X : X = t$ which reduces to true. A formal condition for abstract restriction/projection/existential quantification is as follows: for a pair $\langle A, E \rangle$ being restricted to $\langle A, E_R \rangle$, one requires: $\forall eq(\theta) \in \gamma(E) : \exists eq(\sigma) \in \gamma(E_R)$ such that $A\theta = A\sigma$.

Several researchers, with a background in denotational semantics, have formulated the semantics of logic programs as a function mapping call substitutions into their success substitutions [JoS87]. In this setting, a top down analysis aims at computing that part of the function which is relevant for a given set of initial goals [Win92]. The thus obtained information corresponds closely with the information derived by the approaches using *OLD* semantics.

1.3 From Logic Programming to Constraint Logic Programming

1.3.1 Constraint Logic Programming

Constraint Logic Programming (CLP) is the extension of the Logic Programming paradigm with the constraint solving paradigm. The integration of both paradigms yields a paradigm which is more flexible and more expressive.

Syntactically, the extension introduces constraints in the goal statements and in the bodies of clauses. These constraints are constructed from a so-called signature Σ defining a set of function and predicate symbols and including the $=/2$ predicate. These function and predicate symbols are given a fixed interpretation by means of a so called Σ -structure consisting of a domain $\mathcal{D}$ and an assignment linking the symbols of Σ with the elements of $\mathcal{D}$. The $=/2$ predicate is interpreted as syntactical identity; the other symbols of Σ typically have an interpretation different from the Herbrand interpretation. Constraints are formulas in some first order language over Σ which contains at least all primitive constraints of Σ and is closed under renaming, conjunction and existential quantification.

The simplest example is standard logic programming. Besides the predicate symbol $=/2$, Σ contains a collection of constants and function symbols. The domain $\mathcal{D}$ is the Herbrand universe H , i.e. the set of finite trees and the function

symbols are interpreted as tree constructors. The primitive constraints in L are equations between terms.

Example 1.16 Append in $CLP(H)$

The definition for append is written as⁵:

$$\begin{aligned} \text{append}(X, Y, Z) &\leftarrow X = \text{nil}, Y = Z \quad \square. \\ \text{append}(X, Y, Z) &\leftarrow X = [X' | U], Z = [X' | W] \quad \square \text{ append}(U, Y, W). \end{aligned}$$

Notice that X' is local to the constraint part, so it is existentially quantified. The explicit constraint is $\exists X' : (X = [X' | U], Z = [X' | W])$. $\square$

Another well known example is $CLP(R)$. Σ contains the binary predicate symbols $=$, $<$ and $\leq$, the binary function symbols $+$ and $*$ and the constants 0 and 1. $\mathcal{D}$ is the set of real numbers and the symbols of Σ are interpreted as usual when doing arithmetic. Both domains can be combined, yielding $CLP(H, R)$.

Example 1.17 Listlength in $CLP(H, R)$

listlength can be defined as:

$$\begin{aligned} \text{listlength}(L, N) &\leftarrow L = \text{nil}, N = 0 \quad \square. \\ \text{listlength}(L, N) &\leftarrow L = [X | L'], N = N' + 1 \quad \square \text{ listlength}(L', N'). \end{aligned}$$

Also here, there is a variable local to the constraint part of the body, and a more explicit representation of the constraint is $\exists X : L = [X | L'], N = N' + 1$. $\square$

For more details, the reader is referred to [JaM93] and [Hen91].

1.3.2 Abstracting semantics of CLP

Looking back at the different ways we performed abstract interpretation of logic programs, we observe that the basic entities being abstracted are sets of atoms. In the case of OLD semantics, these sets include non ground atoms. It means we are working with observables similar to computed answer substitutions as formalised in the S -semantics of [FLM89, BGL93]. There it is shown that the sets of atoms representing the meaning of a program are closed under variable renaming. Intuitively, computed answers $X \leftarrow f(Z)$ and $X \leftarrow f(Y)$ for a query $\leftarrow P(X)$ are equivalent. The name of the variable (Z or Y) in the answer depends on the arbitrary renaming of the clauses used in the refutation (on arbitrary addresses in the real implementation). So, back to abstract interpretation one can say that it is advantageous to have that the set of states computed by the concretization function is closed under variable renaming. Failure to do so might cause extra iterations in fixpoint computations.

⁵In this section, the symbol $\square$ is used as a separator between the constraint part and the logic part of a clause.

Example 1.18

Consider a table entry $\langle p(X, Y), \{X = G, Y = f(Z)\} \rangle$ in abstract $OLDT$ or abstract $LSLDT$. A formulation of the concretization function which is more useful than the concretization function following naturally from the setting of example 1.6:

$$\begin{aligned} \gamma(\langle p(X, Y), \{X = G, Y = f(Z)\} \rangle) = \\ \{ \langle p(X, Y), \{X = t, Y = f(Z)\} \rangle \mid t \text{ is ground,} \\ \rho \text{ is a renaming of the variables in the right hand side of the equations} \} \end{aligned}$$

$\square$

The S -semantics have been extended to CLP in [GaL91]; in this extension, the basic entities being manipulated become unit CLP clauses (constrained atoms) of the form $p(X) \leftarrow c \square$. Two constrained atoms $p(X) \leftarrow c_1 \square$ and $p(X) \leftarrow c_2 \square$ are equivalent if they have the same solutions for the variable X .

Example 1.19

$$\begin{aligned} p(X, Y) &\leftarrow X + Y = Z, X - Y = Z, Z = 2 \square \\ p(X, Y) &\leftarrow X + Y = 2, X - Y = 2 \square \\ p(X, Y) &\leftarrow X = 2, Y = 0 \square \end{aligned}$$

These three constrained atoms are equivalent with regard to X and Y . $\square$

The different semantics for LP can easily be reformulated for CLP: one can formulate an operator for the classical fixpoint semantics, or for the S -semantics. Also $OLDT$ and $LSLDT$ can be reformulated. In the latter case, table entries become constrained atoms. While one can consider these constrained atoms as syntactical objects, it is, as in the case of LP, advantageous to consider the equivalence class of semantically equivalent constrained atoms (an implementation freely chooses one of them). This will substantially reduce the number of table entries. Certainly for the elements of the abstract domain, it is recommended to have elements which, under the concretization function, yield sets of constrained atoms which are closed under semantical equivalence.

So, the semantics and frameworks we described in section 1.2 for LP carry over rather straightforwardly to CLP. The main difference being that constraints play the role of substitutions. Typically, one defines a Galois connection between sets of constrained atoms and their abstraction. The order relation on the concrete domain is the subset relation. As explained above, the concretization function should yield a set which is closed under constraint equivalence. The main difficulty in developing applications is that this is harder to achieve than for LP where one only has to cope with variable renaming. For example, in a freeness analysis for solvable systems of linear equations [DJB93], abstraction is conceptually performed on the infinite set of all linear combinations of the given equations. An equation $a_1 X_1 + \dots + a_n X_n = b$ ($a_1 \neq 0, \dots, a_n \neq 0$) is abstracted as $\{X_1, \dots, X_n\}$; so, a system of equations is abstracted as a finite

set of sets of variables. Freeness, whether a variable X_i can still range over the whole set of reals, is indicated by the absence of the singleton $\{X_i\}$ in the abstraction. The concretization function of an abstract element yields a set of solvable systems. Every solvable system in the set is such that its abstraction is a subset of the abstract element.

In some applications, one can use a concretization function which yields sets which are closed under the inverse of entailment, i.e. if a constrained atom $p(X) \leftarrow c \square$ is included, then also all more constrained atoms are included. In LP such domains are known as substitution closed. An example for LP and CLP is definiteness: whether a variable is constrained to at most one value. It is quite obvious that, when a variable is definite for some constraint system C , then it is also definite in stronger systems C' which entail C .

Many authors contributed to the insight regarding the relationship between abstract interpretation for LP and for CLP. A first contribution was by [MaS90]; others are [GBH93, CoF92, GDL92, GDL93].

To finish, we take a closer look at the CLP case of abstract *LSLDT* and formulate the correctness conditions of the abstract operations. Without loss of generality, we assume that clauses are of the form $h \leftarrow b_1, \dots, b_n$ ⁶ where the b_i are either constraints or predicates; moreover, all predicates of the clause are of the form $p(\bar{X})$ where $\bar{X}$ is a shorthand for a sequence of different variables, its length corresponds to the arity of the predicate.

LSLDT consists of subderivations which find their origin either in the initial query or in a program clause. A subderivation is a sequence of goals. In the abstract space, a goal is represented as $\leftarrow b_1, \dots, b_n; c^{\alpha}_{\bar{X}}$ where $b_1, \dots, b_n$ are the operations to be performed (predicates to be called or constraints to be added) and $c^{\alpha}_{\bar{X}}$ is an abstract constraint system over the variables $\bar{X}$ (the abstraction of the constraint store)⁷. The vector $\bar{X}$ contains the variables from the context in which the subderivation is performed, i.e. either the variables from the initial query or from the clause which initiated the subderivation⁸. The abstract goal represents the set of concrete goals $\leftarrow b_1, \dots, b_n; c_{\bar{X}}$ such that $c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{X}})$. Here $c_{\bar{X}}$, the constraint store, is a conjunction of constraints whose free variables are a subset of $\bar{X}$.

The simplest derivation step is when b_1 is a constraint. In this case, the concrete derivation adds b_1 to the constraint store. The new state in the abstract derivation is of the form $\leftarrow b_2, \dots, b_n; c^{\alpha'}_{\bar{X}}$ and the correctness condition for $c^{\alpha'}_{\bar{X}}$ is:

$$\text{if } c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{X}}) \text{ then } c_{\bar{X}} \wedge b_1 \in \gamma(c^{\alpha'}_{\bar{X}}).$$

As this operation is abstracting conjunction, it is called abstract conjunction.

With b_1 a predicate of the form $p(\bar{Y})$ with $\bar{Y} \subseteq \bar{X}$ the step is more involved. In the concrete case, we consider the call $p(\bar{Y})$ with constraint store $\exists_{\bar{Y}} c_{\bar{X}}$ ⁹. If it

⁶So we stop using $\square$ as a separator between constraint and logic part.

⁷so ; is not a separator between constraint part and logic part but between a goal statement and a constraint store; $\square$ will be used for the empty goal statement.

⁸Careful reading will reveal that this is an invariant of the computation scheme.

⁹ $\exists_{\bar{Y}} c_{\bar{X}}$ stands for $\exists x_1, \dots, x_n c$ with $\{x_1, \dots, x_n\} = \text{var}(c) - \bar{Y}$.

is in the table, then the lemmas describing the answers are used to resolve the call, if not, then subderivations are created: for each clause $p(\bar{Z}) \leftarrow d_1, \dots, d_m$ with variables $\bar{U} \supseteq \bar{Z}$, a subderivation $\leftarrow d_1, \dots, d_m; \exists_{\bar{U}}((\exists_{\bar{Y}} c_{\bar{X}}) \wedge (\bar{Z} = \bar{Y}))$ is created. Successful subderivations give rise to lemmas $p(\bar{Z}) \leftarrow c_{\bar{Z}}$ which are then used to solve $p(\bar{Y})$ by lookup. Table lookup derives the state $\leftarrow b_2, \dots, b_n; c_{\bar{X}} \wedge \exists_{\bar{X}}(\bar{Z} = \bar{Y} \wedge c_{\bar{Z}})$.

The abstract computation mimics these operations. The constraint store $c^{\alpha}_{\bar{Y}}$ associated with the call $p(\bar{Y})$ is computed; the correctness condition for $c^{\alpha}_{\bar{Y}}$ is:

$$\text{if } c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{X}}) \text{ then } \exists_{\bar{Y}} c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{Y}}).$$

If this pattern is not in the table then subderivations are created: for each clause $p(\bar{Z}) \leftarrow d_1, \dots, d_m$ with variables $\bar{U} \supseteq \bar{Z}$, a subderivation $\leftarrow d_1, \dots, d_m; c^{\alpha}_{\bar{U}}$ is created. The correctness condition for $c^{\alpha}_{\bar{U}}$ is:

$$\text{if } c_{\bar{Y}} \in \gamma(c^{\alpha}_{\bar{Y}}) \text{ then } \exists_{\bar{U}}(\bar{Z} = \bar{Y} \wedge c_{\bar{Y}}) \in \gamma(c^{\alpha}_{\bar{U}}).$$

Lookup uses a lemma $p(\bar{Z}) \leftarrow c^{\alpha}_{\bar{Z}}$ and derives $\leftarrow b_2, \dots, b_n; c^{\alpha'}_{\bar{X}}$. The correctness condition for $c^{\alpha'}_{\bar{X}}$ is:

$$\text{if } c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{X}}) \text{ and } c_{\bar{Z}} \in \gamma(c^{\alpha}_{\bar{Z}}) \text{ then } c_{\bar{X}} \wedge \exists_{\bar{X}}(c_{\bar{Z}} \wedge \bar{Z} = \bar{Y}) \in \gamma(c^{\alpha'}_{\bar{X}}).$$

Finally, there is the case of a successful subderivation. Let $p(\bar{Z}) \leftarrow d_1, \dots, d_m$ be the clause activated for a call $p(\bar{Y})$ with constraint store $c_{\bar{Y}}$ and let the answer be $\leftarrow \square; c_{\bar{U}}$. The lemma stored as answer for the call pattern is $p(\bar{Z}) \leftarrow \exists_{\bar{Z}} c_{\bar{U}}$. The abstract computation mimics this. We have a call $p(\bar{Y})$ with constraint store $c^{\alpha}_{\bar{Y}}$ and answer $\leftarrow \square; c^{\alpha}_{\bar{U}}$. The obtained lemma is $p(\bar{Z}) \leftarrow c^{\alpha}_{\bar{Z}}$. The correctness condition for $c^{\alpha}_{\bar{Z}}$ is:

$$\text{if } c_{\bar{U}} \in \gamma(c^{\alpha}_{\bar{U}}) \text{ then } \exists_{\bar{Z}} c_{\bar{U}} \in \gamma(c^{\alpha}_{\bar{Z}}).$$

However, only one lemma is allowed for each call pattern; so if there was already a lemma $p(\bar{Z}) \leftarrow c^{\alpha'}_{\bar{Z}}$, then the lub of both lemma's, i.e. $p(\bar{Z}) \leftarrow \text{lub}(c^{\alpha}_{\bar{Z}}, c^{\alpha'}_{\bar{Z}})$ is replacing it. Moreover, all subderivations which were using the old lemma have to be redone, starting from the point where the old lemma was used.

Notice that the top-level derivation terminates with $\leftarrow \square, c_{\bar{X}}$ in the concrete computation and with $\leftarrow \square, c^{\alpha}_{\bar{X}}$ in the abstract computation, with $\bar{X}$ the variables of the initial goal and that $c_{\bar{X}} \in \gamma(c^{\alpha}_{\bar{X}})$.

1.4 Acknowledgements

We are indebted to Anne Mulkers, Veroniek Dumortier, Gerda Janssens, Roberto Giacobazzi and Panagiotis Tsarchopoulos for discussions and comments on drafts.

This work was supported by the "Belgian National Fund for Scientific Research", the "Research Council" of the Katholieke Universiteit Leuven and by the ESPRIT project 5246 PRINCE.

References

- [Bru91] Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. *J. Logic Programming*, 10, (1991) 91-124
- [BGL93] Bossi, A., Gabrielli, M., Levi G., Martelli, M.: The *S*-semantics Approach: Theory and Applications. Draft, submitted
- [CoC92a] Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. *J. Logic and Computation*, 2, (1992) 511-547
- [CoC92b] Cousot, P., Cousot, R.: Abstract Interpretation and Application to Logic Programs. *J. Logic Programming*, 13, (1992) 103-180
- [CoC92c] Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. *Proc. 5th Int. Symp. Programming Language Implementation and Logic Programming*, LNCS, Springer-Verlag, 631, (1992) 269-295
- [CoF92] Codognet, P., Filé, G.: Computations, Abstractions and Constraints in Logic Programs. *Proc. 4th Int. Conf. on Programming Languages*, Oakland, 1992
- [CoH78] Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. *Conference Record of the 5th ACM Symp. on Principles of Programming Languages*, ACM Press, 1978, 84-97
- [CVH91] Le Charlier, B., Van Hentenryck, P.: A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract). *Proc. 8th Int. Conf. on Logic Programming*, MIT Press, 1991, 64-78
- [CVH92] Le Charlier, B., Van Hentenryck, P.: Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *Proc. 4th IEEE International Conference on Computer Languages*, San Francisco, 1992
- [DJB93] Dumortier, V., Janssens, G., Bruynooghe, M., Codish, M.: Freeness Analysis in the Presence of Numerical Constraints. *Proc. 5th Int. Conf. on Logic Programming*, MIT Press, 1993, 100-115
- [FLM89] Falaschi, M., Levi, G., Martelli, M., Palamidessi, C.: Declarative Modelling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science*, 69, (1989) 289-318
- [GaL91] Gabrielli, M., Levi, G.: Modelling Answer Constraints in Constraint Logic Programs. *Proc. 8th Int. Conf. Logic Programming*, MIT Press, 1991, 301-315
- [GBH93] García de la Banda, M., Hermenegildo, M.: A Practical Approach to the Global Analysis of CLP Programs. *Proc. 1993 Int. Logic Programming Symp.*, MIT Press, 1993

- [GDL92] Giacobazzi, R., Debray, S., Levi, G.: Generalised Semantics and Abstract Interpretation for Constraint Logic Programs. Proc. Int. Conf. on Fifth generation Computer Systems, Tokyo, 1992
- [GDL93] Giacobazzi, R., Debray, S., Levi, G.: Generalised Semantics and Abstract Interpretation for Constraint Logic Programs. Preliminary Report, April 1993
- [Hen91] Van Hentenryck, P.: Constraint Logic Programming. Knowledge Engineering Review, 6, (1991) 151-194
- [KaK93] Kanamori, T., Kawamura, T.: Abstract Interpretation Based on OLD Resolution. J. Logic Programming 15, (1993) 1-30
- [JaM93] Jaffar, J., Maher, M.: Constraint Logic Programming, a Survey. IBM T.J. Watson Research Center, Draft, submitted
- [JoS87] Jones, N., Sondergaard, H.: A Semantics-Based Framework for the Abstract Interpretation of PROLOG. In S. Abramsky and C. Hankin (eds.), Abstract Interpretation for Declarative Languages, Ellis Horwood, 1987, 123-142
- [Karr76] Karr, M.: Affine Relationships among Variables of a Program. Acta Inform., 6, (1976) 133-151
- [Llo87] Lloyd, L.: Foundations of Logic Programming. Springer-Verlag, Berlin, 1987
- [LMM88] Lassez, J.-L., Maher, M., Marriott, K.: Unification Revisited. In J. Minker (ed.), Foundations of Deductive Databases and Logic Programming, Morgan-Kaufmann, 1988, 587-625
- [MaM82] Martelli, A., Montanari, U.: An Efficient Unification Algorithm. ACM Transactions on Programming Languages and Systems, 4, (1982) 258-282
- [MaS90] Marriott, K., Sondergaard, H.: Analysis of Constraint Logic Programs. Proc. 1990 North American Conference on Logic Programming, MIT Press, 1990, 532-547
- [MuH92] Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. J. Logic Programming, 13, (1992) 315-374
- [TaS86] Tamaki, H., Sato, T.: OLD Resolution with Tabulation. Proc. 3rd Int. Conf. on Logic Programming, Springer-Verlag, 1986, 84-98
- [VDS91] Verschaetse, K., De Schreye, D.: Deriving Termination Proofs for Logic Programs. Proc. 8th Int. Conf. Logic Programming, MIT Press, 1991, 301-315
- [VDS92] Verschaetse, K., De Schreye, D.: Derivation of Linear Size Relations by Abstract Interpretation. Proc. 5th Int. Symp. Programming Language Implementation and Logic Programming'92, LNCS, Springer-Verlag, 631, (1992) 296-310
- [Vie89] Vieille, L.: Recursive Query Processing: The power of Logic. Theoretical Computer Science, 69, (1989) 1-53
- [Win92] Winsborough, W.: Multiple Specialisation Using Minimal-Function Graph Semantics. J. Logic Programming, 13, (1992) 259-290