Electronic Notes in Theoretical Computer Science 42 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume42.html` 17 pages

Precise Goal-Independent Abstract Interpretation of Constraint Logic Programs

Peter Schachte [1]

The University of Melbourne
Victoria 3010, Australia

Abstract

We present a goal-independent abstract interpretation framework for pure constraint logic programs, and prove the sufficiency of a set of conditions for abstract domains to ensure that the analysis will never lose precision. Along the way, we formally define pure constraint logic programming systems, give a formal semantics that is independent of the actual constraint domain, and formally define the maximally precise abstraction of a pure constraint logic program.

1 Introduction

Abstract interpretation [1] is the process of mimicking the formal semantics of a program — interpreting the program — using an abstraction of the data used by the real program. By basing our analysis on the formal semantics of the program, we gain guarantees of the correctness of our results, and by using an abstraction of the real data, we often gain a guarantee of termination.

The central idea of abstract interpretation is to *approximate* the actual data of a program. An approximation of a program state, which we call an *abstract value*, will usually approximate more than a single program state. For example, we might approximate an integer by whether it is even or odd, or whether it is smaller, greater or equal to zero. Often, however, we will not be able to choose a single one of these abstractions. For example, we may know an integer variable will be either 0 or 3; in this case it could be either 0 or greater than zero. Therefore, it is not sufficient to have a set of approximate values $\{<0, =0, >0\}$. A useful set of approximations might include $\{<0, \leq 0, = 0, \neq 0, \geq 0, >0\}$. It must also include an abstraction to indicate no information, or perfect uncertainty. This will usually be denoted $\top$. It is also convenient to include another abstraction to indicate that no value is possible, to handle

[1] Email: `schachte@cs.mu.oz.au`

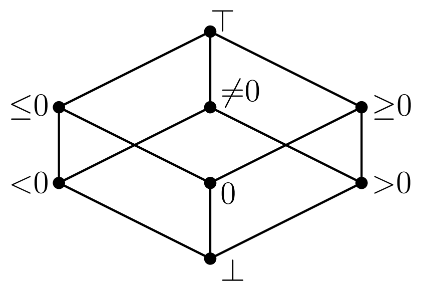

Fig. 1. Hasse diagram for the signs abstract domain for a single variable

cases of unreachable, failing, (infinitely) looping, or error-causing code. This value is usually denoted $\perp$.

Another way to think about this is that we are partitioning $\mathbb{Z}$ into three distinct sets, each corresponding to an element of $\{<0, =0, >0\}$. Then for any *set* of integers we can find a subset of $\{<0, =0, >0\}$ that covers all the integers in the given set. Thus the set of abstract values we are interested in is the powerset of the set of signs, $\mathcal{P}(\{<0, =0, >0\}) = \{\varnothing, \{<0\}, \{=0\}, \{>0\}, \{<0, =0\}, \{<0, >0\}, \{=0, >0\}, \{<0, =0, >0\}\}$. These abstract values will be easier to work with if they are given more convenient names. In the same order, we will use the names $\{\perp, <0, =0, >0, \leq 0, \neq 0, \geq 0, \top\}$.

Clearly we can approximate any set of values by $\top$, but some approximations are better than others, and we would always like to choose the most precise approximation. To continue with our example, if a program variable could take on any of the values $1, 3, 5, 7, \ldots$, we would prefer to approximate it as >0, even though ≥ 0 or $\top$ would also be correct, because ≥ 0 and $\top$ describe a larger number of concrete values than >0. This gives us a means of comparing approximations, and makes our set of abstract values a poset. Furthermore, every subset of our set of approximations has a least upper bound and a greatest lower bound, which means that we have a complete lattice, as depicted in the Hasse diagram in Figure 1 . If we think of our set of abstract values as a power set, then we can use subset ordering to guarantee us a complete lattice. Such a set of possible approximations, together with its ordering relation, is called an *abstract domain*.

Once we have found an abstract domain, we will want to be precise about the meanings of its elements. For this, we want a *concretization function*

$$\gamma : \mathsf{Abst} \to \mathcal{P}(\mathsf{Conc}).$$

We use the name Conc to refer to the concrete domain, in this example, the set of integers; Abst refers to the abstract domain, in this case $\{\perp, <0, =0, >0, \leq 0, \neq 0, \geq 0, \top\}$.

Note that γ maps abstractions to *sets* of concrete values, because there will always be more than a single concrete value corresponding to some abstract values (otherwise the abstract domain is not very abstract). We will also need a function to abstract the concrete values appearing in the program to be

$$\begin{tikzcd}
\alpha\ \{c\} \arrow[r, "F_\alpha"] & F_\alpha\ \alpha\ \{c\} = \alpha\, F\, c \\
c \arrow[u, "\alpha"] \arrow[r, "F"'] & F\ c \arrow[u, "\alpha"']
\end{tikzcd}$$

Fig. 2. Correctness condition for function approximation

analyzed. We define an *abstraction function* as

$$\alpha : \mathcal{P}(\mathsf{Conc}) \to \mathsf{Abst}.$$

That is, given a *set* of concrete values, α yields the appropriate (most precise) abstraction for that set. This should be the least upper bound of the abstractions we expect for each element of the set. Note that α must be monotonic because adding to a set of concrete values to be abstracted should never result in a more precise abstraction. If this were to happen, then the more precise abstraction would certainly apply to the smaller set. Conversely, we require that γ must be monotonic because a less precise abstraction should never describe a smaller set of concrete values. Further, we will require that

$$\forall C \subseteq \mathsf{Conc} : C \subseteq \gamma\ (\alpha\ C), \text{ and}$$
$$\forall a \in \mathsf{Abst} : \alpha\ (\gamma\ a) \sqsubseteq a$$

The first of these inequalities guarantees that abstracting and then concretizing a set of values doesn't "lose" any values, while the second assures us that concretizing and then abstracting an abstract value won't lose any precision. This means that α and γ form a Galois connection. Ideally, we would like to have

$$\forall a \in \mathsf{Abst} : \alpha\ (\gamma\ a) = a$$

(meaning that α and γ form a Galois insertion), but we shall not require it.

Next we need to determine how to abstract concrete operations on the data we are interested in. That is, for each concrete operation $F : \mathsf{Conc} \to \mathsf{Conc}$, we must find an abstract operation $F_\alpha : \mathsf{Abst} \to \mathsf{Abst}$ that faithfully approximates it. Fortunately, our α and γ functions make clear how to do this: we must require that

$$\forall c \in \mathsf{Conc} : F\ c \in \gamma(F_\alpha\ (\alpha\{c\})).$$

This condition is often expressed by the diagram in Figure 2 . This is only a correctness constraint, however; for optimality we would also like there to be no F'_α satisfying this constraint and also satisfying

$$\exists a \in \mathsf{Abst} : F'_\alpha\ a \sqsubseteq F_\alpha\ a.$$

The balance of this paper is arranged as follows. Section 2 presents a denotational semantics for constraint logic programs which is independent of any particular constraint domain. It also specifies what properties we expect a constraint logic programming system to exhibit. In Section 3 we define an abstract interpretation framework, specify the properties we expect of an abstract domain, and prove that our abstract interpretation framework will always produce a maximally precise analysis of any program providing that the abstract domain used exhibits the properties we require. Section 4 describes related work, and Section 5 presents our conclusions.

2 The Semantics of Constraint Logic Programs

To be confident of the correctness of our analyses of the behavior of constraint logic programs, we need to specify exactly how programs behave.

In this paper, we will concern ourselves with only the goal independent analysis of pure constraint logic programs. This decision permits us to develop a semantics, and an analysis, that is independent of implementation technology. For a discussion of the issues of this paper related to goal-dependent analysis, see Schachte [2]. Our semantics is also independent of the choice of a constraint domain.

We assume we are given the following denumerable disjoint sets:

Var the set of all variables;

Fn the set of all constructor functions;

PCon the set of all primitive constraint constructors;

Pred the set of all atom constructors.

Since equality is an essential part of all logic programming, it must be part of all constraint domains [3]. Therefore, PCon must include $=$, the special equality constraint.

From these sets we define Term to be the union of Var and the set of all terms that can be constructed with functors from Fn and arguments from Term. Given $t, t' \in \mathsf{Term}$ and $v \in \mathsf{Var}$, we denote by $t\,[t'/v]$ the unique result of replacing all occurrences of v in t by t'. We define $\mathsf{vars} : \mathsf{Term} \to \mathsf{Var}$ to yield the set of all variables appearing syntactically in a term.

Note that $t\,[t'/v]$ has some important properties which we will need. Firstly, this operation replaces *all* occurrences of v, so that if $v \notin \mathsf{vars}(t')$, then $v \notin \mathsf{vars}(t\,[t'/v])$. Also note that for any variable v', if $v' \notin \mathsf{vars}(t)$ then $t = (t\,[v'/v]\,[v/v'])$. Finally, if two substitutions are independent, they may be applied in either order; that is, if $v' \notin t$, $v \notin t'$, and $v \neq v'$, then $t''\,[t/v]\,[t'/v'] = t''\,[t'/v']\,[t/v]$.

Prim is the set of all primitive constraints formed with constructors from PCon and arguments from Term, and similarly Atom is the set of all atoms formed with constructors from Pred and arguments from Term. Lit is the set of literals $\mathsf{Atom} \cup \mathsf{Prim}$. Each element of $\mathsf{Body} = \mathcal{P}_{\mathrm{f}}(\mathsf{Lit})$, that is, the set of finite sets of elements from Lit, representing finite conjunctions of literals.[2]

[2] Since we consider only goal-independent analysis of pure contstraint logic programs, and

Clause is the set of clauses $H \leftarrow B$ where $H \in \mathsf{Atom}$ and $B \in \mathsf{Body}$. Finally, $\mathsf{Program} = \mathcal{P}_{\mathrm{f}}(\mathsf{Clause})$ is the set of all finite programs composed of clauses from Clause. As a convenience, we extend $=$ to $\mathsf{Atom} \times \mathsf{Atom}$ as an abbreviation for the equivalence of the predicate constructors and the pairwise equivalence of the arguments.

Next we define Con to be the set of all possibly existentially quantified finite conjunctions of primitive constraints. We view an element of $\mathcal{P}(\mathsf{Con})$ as describing a number of alternative constraints, that is, a disjunction of constraints.

We extend our definition of vars to cover Prim, Atom, Lit, Con, and Clause in the obvious syntactic way.

There are several properties which we require of any constraint logic programming system we are to analyze.

Axiom 2.1 *For $c, c', c'' \in \mathsf{Con}$, $t, t' \in \mathsf{Term}$, and $v, v' \in \mathsf{Var}$:*

(i) *Conjunction must be commutative, associative, and absorptive* (i.e., $\ulcorner c \wedge c' \urcorner \equiv \ulcorner c' \wedge c \urcorner$, $\ulcorner c \wedge c \urcorner \equiv c$, *and* $\ulcorner c \wedge (c' \wedge c'') \urcorner \equiv \ulcorner (c \wedge c') \wedge c'' \urcorner$)[3]

(ii) true *must be an identity, and* false *an annihilator for conjunction* ($\ulcorner \mathsf{true} \wedge c \urcorner \equiv \ulcorner c \wedge \mathsf{true} \urcorner \equiv c$, $\ulcorner \mathsf{false} \wedge c \urcorner \equiv \ulcorner c \wedge \mathsf{false} \urcorner \equiv \mathsf{false}$)

(iii) $\ulcorner \exists v : \mathsf{true} \urcorner \equiv \mathsf{true}$ *and* $\ulcorner \exists v : (c \wedge c') \urcorner \equiv \ulcorner c \wedge \exists v : c' \urcorner$ *when* $v \notin \mathsf{vars}(c)$;

(iv) $\ulcorner \exists v : \exists v' : c \urcorner \equiv \ulcorner \exists v' : \exists v : c \urcorner$;

(v) $\ulcorner \exists v : c \urcorner \equiv \ulcorner \exists v' : c' \urcorner$ *when* $v' \notin \mathsf{vars}(c) \wedge c' \equiv c\,[v'/v]$;

(vi) *The required equality constraint $=$ must be an equivalence relation* ($\ulcorner t = t \urcorner \equiv \mathsf{true}$, $\ulcorner t = t' \urcorner \equiv \ulcorner t' = t \urcorner$, *and* $\ulcorner t = t' \wedge t' = t'' \urcorner \equiv \ulcorner t = t'' \urcorner$)

(vii) $\ulcorner \exists v : v = t \wedge c \urcorner \equiv c\,[t/v]$ *when* $v \notin \mathsf{vars}(t)$;

(viii) $\ulcorner \exists v : (v = t \wedge c \wedge c') \urcorner \equiv \ulcorner (\exists v : v = t \wedge c) \wedge (\exists v : v = t \wedge c') \urcorner$ *when* $v \notin \mathsf{vars}(t)$

□

We extend the conjunction operation $\wedge$ on Con to $\wedge : \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$ as a *cross-conjunction*, that is, for $S_1, S_2 \subseteq \mathsf{Con}$,

$$S_1 \wedge S_2 = \{s_1 \wedge s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\} \setminus \{\mathsf{false}\}.$$

(Because false is not a meaningful constraint in that it can never be satisfied, it is convenient for cross-conjunction to remove it.) We naturally extend this to a unary function $\bigwedge : \mathcal{P}_{\mathrm{f}}(\mathcal{P}(\mathsf{Con})) \to \mathcal{P}(\mathsf{Con})$ as repeated cross-conjunction. We also extend existential quantification to $\mathcal{P}(\mathsf{Con})$ in a similar way: for $S \subseteq$

we shall require that conjunction be commutative, associative, and absorptive, sets of literals are an adequate representation for clause bodies.

[3] We distinguish elements of a program being analyzed from the elements of the analysis framework being presented using *Quine corners* [4], written $\ulcorner \cdot \urcorner$. Quine corners are not quotes; the enclosed material may include variables, which are to be interpreted as such.

$\mathsf{Con}, v \in \mathsf{Var}$, we define

$$\exists v : S = \{(\exists v : s) \mid s \in S\};$$

we further extend $\exists$ to handle quantification of possibly infinite sets of variables in the natural way.

It will be convenient to have a notation for performing many substitutions at once, which we gain through the extension of our notation for substitution to apply to *sequences* of variables and terms. We denote by $t\,[\boldsymbol{t}/\boldsymbol{v}]$ the pairwise substitution of terms from $\boldsymbol{t}$ for variables from $\boldsymbol{v}$ in t, where $\boldsymbol{t}$ is a sequence of terms, $\boldsymbol{v}$ is a sequence of variables of the same length, no variable appears more than once in $\boldsymbol{v}$, and $\boldsymbol{v}$ and $\mathsf{vars}(\boldsymbol{t})$ are disjoint. We will abuse notation by applying set operations to sequences, and understand these operations to apply to the set of elements of the sequence. Our need for sequences is small enough that it is not important that we be formal about them; we only note that for any set s it is possible to arbitrarily choose a sequence $\boldsymbol{s}$ such that $\boldsymbol{s}$ contains all and only the elements of s, without repetition. Note that our observations about $t\,[t'/v]$ apply to sequences as well, as long as the sequences contain no repeated elements. In particular, $t\,[\boldsymbol{v}'/\boldsymbol{v}]\,[\boldsymbol{v}/\boldsymbol{v}'] = t$ whenever $\boldsymbol{v}$ and $\boldsymbol{v}'$ are disjoint, and $\boldsymbol{v}'$ and $\mathsf{vars}(t)$ are disjoint.

Giacobazzi *et al.* [5] identify a number of properties that a *constraint system* must exhibit in order to be "sensible," that is, in order to behave in a manner we would expect of a constraint logic programming system. The following theorem proves that $\mathcal{P}(\mathsf{Con})$, together with the operations $\wedge, \cup$, and $\exists$, the elements $\{\mathsf{true}\}$ and $\varnothing$, and at least the primitive constraint $=$, meet these requirements.

Theorem 2.2 *For every $V, V' \subseteq \mathsf{Var}, v \in \mathsf{Var}, C, C' \subseteq \mathsf{Con}$, and $t, t', t'' \in \mathsf{Term}$, we have the following:*

(i) $\wedge$ *distributes over finite and infinite unions;*

(ii) $\wedge$ *is associative and has an identity* $\{\mathsf{true}\}$*;*

(iii) $\cup$ *is associative and has an identity* $\varnothing$*;*

(iv) $\cup$ *is commutative and absorptive;*

(v) $\varnothing$ *is an annihilator for* $\wedge$*;*

(vi) *for any possibly infinite set* $S \subseteq \mathcal{P}(\mathsf{Con})$*,* $\bigcup S$ *exists and is unique;*

(vii) $(\exists V : \varnothing) \equiv \varnothing$*;*

(viii) $\exists V : (C \wedge \exists V : C') \equiv \exists V : ((\exists V : C) \wedge C') \equiv (\exists V : C) \wedge (\exists V : C')$*;*

(ix) $\exists V : \exists V' : C \equiv \exists (V \cup V') : C$*;*

(x) $\exists V$ *distributes over finite and infinite unions;*

(xi) $\{\ulcorner t = t \urcorner\} \equiv \{\mathsf{true}\}$*;*

(xii) $\{\ulcorner t = t' \urcorner\} \equiv \{\ulcorner t' = t \urcorner\}$*;*

(xiii) $\exists v : (\ulcorner v = t \urcorner \wedge \ulcorner t' = t'' \urcorner) \equiv (\ulcorner t' = t'' \urcorner)\,[t/v]$ *when* $v \notin t$*;*

(xiv) $\ulcorner\exists v : (v = t \wedge (C \wedge C'))\urcorner \equiv \ulcorner\exists v : (v = t \wedge C) \wedge (\exists v : (v = t \wedge C'))\urcorner$ *when* $v \notin$ vars(t).

$\square$

Giacobazzi *et al.* also require that

(xv) $(C \cup \exists V : C) \equiv \exists V : C$,

but we do not wish to require this. To do so would prevent us from collecting extralogical information about predicates that may become important in the analysis of programs. For example, given the Prolog program

```
p(1).
p(X).
```

we would like our semantics to conclude that there are two solutions for `p/1`, one indicating that `1` is a solution and the other that any term is a solution. Including (xv) would only allow the more general solution, thwarting analyses which would need to know that `1` is also a solution. To extend this example, an analysis for definite freeness would need to know that even if `p/1` is invoked with its argument unbound, it can succeed with its argument bound. A determinacy analysis would need to know that `p/1` may have two solutions, and so would not be determinate when its argument is unbound. We do not discuss such analyses in this paper but we do want our framework to allow them.

We require one further function:

$$\text{rename} : \text{Clause} \to \mathcal{P}(\text{Var}) \to \text{Clause}$$

produces a variant of the input clause that has no variables in common with the given set of variables. This enforces the rule that the scope of a variable is limited to the clause it occurs in.

We specify our semantic domain **Den** as

$$\textbf{Den} = \text{Atom} \to \mathcal{P}(\text{Con}).$$

It is ordered pointwise. The least denotation ($\perp$) maps all atoms to the empty set.

Definition 2.3 (Program semantics) *The denotation of a program is given by the function* $\mathbf{P}^{\text{sem}}$*, which we define in terms of the auxiliary functions* $\mathbf{C}^{\text{sem}}$ *and* $\mathbf{L}^{\text{sem}}$*:*

$$\begin{aligned} \mathbf{P}^{\text{sem}} &: \text{Program} \to \textbf{Den} \\ \mathbf{C}^{\text{sem}} &: \text{Clause} \to \textbf{Den} \to \textbf{Den} \\ \mathbf{L}^{\text{sem}} &: \text{Lit} \to \textbf{Den} \to \mathcal{P}(\text{Con}) \end{aligned}$$

These functions are defined as follows:

$$\mathbf{P}^{\text{sem}}\ P = \mathsf{lfp}\ \left(\bigsqcup_{C\in P} \mathbf{C}^{\text{sem}}\ C\right)$$

$$\mathbf{C}^{\text{sem}}\ C\ d\ A = \exists V\ \left((\mathbf{L}^{\text{sem}}\ \ulcorner H = A\urcorner\ d) \wedge \bigwedge_{L\in B} \mathbf{L}^{\text{sem}}\ L\ d\right)$$
$$\textsf{\textit{where}}\ \ulcorner H \leftarrow B\urcorner = \mathsf{rename}\ C\ \ \mathsf{vars}(A)$$
$$\textsf{\textit{and}}\ V = \mathsf{vars}(H) \cup \mathsf{vars}(B)$$

$$\mathbf{L}^{\text{sem}}\ L\ d = \begin{cases} \varnothing & \textsf{\textit{when}}\ \ L \in \mathsf{Prim} \wedge L \equiv \mathsf{false} \\ \{L\} & \textsf{\textit{when}}\ \ L \in \mathsf{Prim} \wedge L \not\equiv \mathsf{false} \\ d\ L & \textsf{\textit{when}}\ \ L \in \mathsf{Atom} \end{cases}$$

□

Intuitively, we define the semantics of a program as the least fixed point of the combination of the semantics of the clauses of the program. Thus the semantics is defined by specifying how, given the set of constraints that can result from n derivation steps, we can compute the constraints resulting from $n+1$ derivation steps. The semantics of an individual clause $H \leftarrow B$ is the function which maps an atom A to the conjunction of the equality constraint $H = A$ and the literals in the body of the clause, and projects away all the variables in the clause (leaving only the variables in the invocation). The semantics of a literal L depends upon whether it is an atom or a primitive constraint. The semantics of an atom is determined by the given denotation function (i.e., the result of the previous derivation step), while the semantics of a primitive constraint is just that constraint as a singleton set (we filter out false constraints because they cannot be satisfied).

3 Abstract Interpretation of Constraint Logic Programs

In abstract interpretation, we wish to find an abstract version of the semantics of a program. In fact, we really wish to abstract the semantics twice: we abstract the meaning of the program to be given in terms of some abstract domain rather than the concrete domain, giving us an abstract interpreter. Then we abstract the choice of abstract domain from this, yielding an abstract interpretation *framework*. The focus of this paper is the presentation of an abstract interpretation framework and the establishment of a set of sufficient conditions for this framework to always produce maximally precise analyses, according to the following definition of "maximally precise."

Definition 3.1 (Precise approximation) *Given an abstract domain* ACon *and an abstraction function* $\alpha : \mathcal{P}(\mathsf{Con}) \to \mathsf{ACon}$*, we say that* $a \in \mathsf{ACon}$ precisely approximates $C \subseteq \mathsf{Con}$*, and write* $a\ \mathsf{appr}_\alpha\ C$*, as follows:*

$$a\ \mathsf{appr}_\alpha\ C \leftrightarrow \alpha\, C = a.$$

For any set S, we extend this relation to functions $F : S \rightarrow \mathsf{ACon}$ and $G : S \rightarrow \mathcal{P}(\mathsf{Con})$ in the natural way:

$$F \;\mathsf{appr}_\alpha\; G \leftrightarrow \forall s \in S : (F\, s) \;\mathsf{appr}_\alpha\; (G\, s).$$

We further extend this relation to functions $F : \mathsf{ACon} \rightarrow \mathsf{ACon}$ and $G : \mathcal{P}(\mathsf{Con}) \rightarrow \mathcal{P}(\mathsf{Con})$ as follows:

$$F \;\mathsf{appr}_\alpha\; G \leftrightarrow \forall a \in \mathsf{ACon}, C \subseteq \mathsf{Con} : (a \;\mathsf{appr}_\alpha\; C \rightarrow (F\, a) \;\mathsf{appr}_\alpha\; (G\, C)).$$

□

Now we specify what we require of an abstraction.

Definition 3.2 (Abstraction) *An* abstraction *comprises the following:*

- *an* abstract domain ACon*, which is a complete lattice ordered by $\sqsubseteq$, and which has $\sqcap, \sqcup, \bot$, and $\top$, as meet, join, bottom, and top, respectively. This lattice is ordered by information content; we follow the usual convention in the abstract interpretation literature and put more information (greater certainty)* lower *in the lattice.*[4]
- *an abstract conjunction function $\curlywedge : \mathsf{ACon} \rightarrow \mathsf{ACon} \rightarrow \mathsf{ACon}$.*
- *a* projection function $\mathsf{project} : \mathcal{P}(\mathsf{Var}) \rightarrow \mathsf{ACon} \rightarrow \mathsf{ACon}$*. This function is an abstraction of existential quantification*
- *for each primitive constraint $c \in \mathsf{Prim}$, an abstract constraint, which we denote c_α. Recall that we always require $=$ to be a primitive constraint, so there must always be an $=_\alpha$.* □

We find it convenient to follow Nielson [6] in characterizing our abstract domain in terms of a representation function. We will define the needed abstraction and concretization functions in terms of the representation function below.

Definition 3.3 *We define a* representation function $\beta : \mathsf{Con} \rightarrow \mathsf{ACon}$ *which gives a maximally precise*[5] *abstraction for each concrete conjunction of constraints as follows:*

$$\beta\ c = \begin{cases} c_\alpha & \textit{when}\ \ c \in \mathsf{Prim} \\ \mathsf{project}\ \ v\ (\beta\ c') & \textit{when}\ \ c = \exists v : c' \\ (\beta\ c')\ \curlywedge\ (\beta\ c'') & \textit{when}\ \ c = c' \wedge c'' \end{cases}$$

□

[4] The reason for this convention is that the concrete domain uses a standard subset ordering, which puts larger sets above smaller ones, and larger sets of solutions are usually abstracted to less certainty about the properties exhibited by *all* solutions.

[5] Ideally, the abstract domain would have a unique most precise abstraction for each concrete conjunction of constraints, but we do not require this.

Now we may specify sufficient conditions for an abstract domain to guarantee that our analysis will always produce a precise analysis.

Definition 3.4 (Precise abstraction) *A* precise abstraction *is an abstraction which satisfies the following constraints:*

- *The abstract conjunction function* must *precisely approximate conjunction. That is,*
$$\forall c, c' \in \mathsf{Con} : \beta\ (c \wedge c') = (\beta\ c) \mathbin{\hat{\wedge}} (\beta\ c')$$
must hold. We further require that $\hat{\wedge}$ distributes over finite and infinite joins, that is, for all (possibly infinite) $A, A' \subseteq \mathsf{ACon}$,
$$(\bigsqcup A) \mathbin{\hat{\wedge}} (\bigsqcup A') = \bigsqcup \{a \mathbin{\hat{\wedge}} a' \mid a \in A \wedge a' \in A'\}.$$
These requirements of $\hat{\wedge}$ also in fact create requirements on the abstract primitive constraints.
- *The abstract projection function must precisely approximate existential quantification (*$\mathsf{project}\ \mathsf{appr}_\alpha\ \exists$*). That is,*
$$\forall V \subseteq \mathsf{Var}, c \in \mathsf{Con} : \mathsf{project}\ V\ (\beta\ c) = \beta\ (\ulcorner \exists V : C \urcorner).$$
In particular, this means that all of the following must hold:
$$\forall C_\alpha \in \mathsf{ACon}, V \subseteq \mathsf{Var} : \mathsf{vars}(C_\alpha) \cap V = \varnothing \rightarrow \mathsf{project}\ V\ C_\alpha = C_\alpha;$$
$$\forall V \subseteq \mathsf{Var} : \mathsf{project}\, V \bot = \bot;$$
$$\forall C_\alpha \in \mathsf{ACon}, V \subseteq \mathsf{Var} : \mathsf{vars}(\mathsf{project}\ V\ C_\alpha) \cap V = \varnothing$$

□

Now we may define our abstraction and concretization functions in terms of our representation function.

Definition 3.5 *From the representation function β, we define the needed concretization function $\gamma : \mathsf{ACon} \rightarrow \mathcal{P}(\mathsf{Con})$ and abstraction function $\alpha : \mathcal{P}(\mathsf{Con}) \rightarrow \mathsf{ACon}$ as follows:*

$$\gamma\ a = \{c \in \mathsf{Con} \mid \beta\ c \sqsubseteq a\}$$
$$\alpha\ C = \bigsqcup_{c \in C} \beta\ c$$

□

This characterization of α and γ give us the following result:

Theorem 3.6 *α and γ form a Galois connection, with α the lower adjoint and γ the upper.* □

Now we define an abstract semantics to be a function

$$\mathbf{ADen} = \mathsf{Atom} \rightarrow \mathsf{ACon}.$$

Finally we may define the abstract semantics of a program. Naturally, the definition is written subject to the choice of a precise abstraction, as specified in Definition 3.4 . All of the constructs of the abstraction are notionally parameters to the abstract semantic functions below, but we do not specify them as such to keep the definition manageable.

Definition 3.7 (Abstract semantic function) *We specify the* abstract semantics *of a program as the result of the (goal independent) abstract semantic function* $\mathbf{P}_\alpha^{\mathrm{sem}}$*, which we define in terms of the auxiliary functions* $\mathbf{C}_\alpha^{\mathrm{sem}}$ *and* $\mathbf{L}_\alpha^{\mathrm{sem}}$*:*

$$\begin{aligned}\mathbf{P}_\alpha^{\mathrm{sem}} &: \mathsf{Program} \to \mathbf{ADen}\\ \mathbf{C}_\alpha^{\mathrm{sem}} &: \mathsf{Clause} \to \mathbf{ADen} \to \mathbf{ADen}\\ \mathbf{L}_\alpha^{\mathrm{sem}} &: \mathsf{Lit} \to \mathbf{ADen} \to \mathsf{ACon}\end{aligned}$$

These functions are defined as follows:

$$\mathbf{P}_\alpha^{\mathrm{sem}}\ P = \mathsf{lfp}\left(\bigsqcup_{C\in P} \mathbf{C}_\alpha^{\mathrm{sem}}\ C\right)$$

$$\mathbf{C}_\alpha^{\mathrm{sem}}\ C\ a\ A = \mathsf{project}\, V\left(\left(H =_\alpha A\right) \mathbin{\overset{\alpha}{\wedge}} \bigwedge_{L\in B}^{\alpha} \mathbf{L}_\alpha^{\mathrm{sem}}\ L\ a\right)$$
$$\textit{where } \ulcorner H \leftarrow \vec{B}\urcorner = \mathsf{rename}\ C\ \ \mathsf{vars}(A)$$
$$\textit{and } V = \mathsf{vars}(H) \cup \mathsf{vars}(B)$$

$$\mathbf{L}_\alpha^{\mathrm{sem}}\ L\ a = \begin{cases} L_\alpha & \textit{when } L \in \mathsf{Prim}\\ a\ L & \textit{when } L \in \mathsf{Atom}\end{cases}$$

□

Given this, we wish to show that $\mathbf{P}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{P}^{\mathrm{sem}}$, but first we must prove a theorem and a lemma.

Theorem 3.8 *For any sets* $C, C' \subseteq \mathsf{Con}$*,*

$$\alpha\ (C \wedge C') = (\alpha\ C) \mathbin{\overset{\alpha}{\wedge}} (\alpha\ C')$$

□

Lemma 3.9 *The* appr_α *relation on* $\mathcal{P}(\mathsf{Con}) \times \mathsf{ACon}$*, ordered componentwise, is admissible for fixed point induction.* □

Now we are equipped to prove the main result of this section: that the abstract semantics given in Definition 3.7 , when applied to any abstraction satisfying Definition 3.4 , will always yield the most precise abstraction of any given program.

Theorem 3.10 $\mathbf{L}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{L}^{\mathrm{sem}}$*,* $\mathbf{C}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{C}^{\mathrm{sem}}$*, and* $\mathbf{P}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{P}^{\mathrm{sem}}$*.*

Proof. First we prove $\mathbf{L}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{L}^{\mathrm{sem}}$. Choose an arbitrary $L \in \mathsf{Lit}$ and $d \in$ **Den**, and let $a = \alpha \circ d$. If $L \in \mathsf{Prim}$, then we must show that $\alpha\,\{L\}\ \mathsf{appr}_\alpha\ \{L\}$, which obviously holds. If $L \in \mathsf{Atom}$, then we must show that $a\,L\ \mathsf{appr}_\alpha\ d\,L$, but since $a = \alpha \circ d$, this is obvious, too.

Now we show that since $\mathbf{L}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{L}^{\mathrm{sem}}$, we also have $\mathbf{C}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{C}^{\mathrm{sem}}$. Choose an arbitrary $C \in \mathsf{Clause}$, $d \in \mathbf{Den}$, and $A \in \mathsf{Atom}$, and let $\ulcorner H \leftarrow B \urcorner =$ $\mathsf{rename}\ C\ \ \mathsf{vars}(A)$ and $V = \mathsf{vars}(H) \cup \mathsf{vars}(B)$. We must show that

$$\mathsf{project}\,V\ \left((H {=_\alpha} A)\ \mathbin{\overset{\alpha}{\wedge}}\ \bigwedge_{L\in B}^{\alpha} \mathbf{L}_\alpha^{\mathrm{sem}}\ L\ a \right) \mathsf{appr}_\alpha\ \exists V\ \left(H = A \wedge \bigwedge_{L\in B} \mathbf{L}^{\mathrm{sem}}\ L\ d \right)$$

Since we require that $\mathsf{project}\ \ \mathsf{appr}_\alpha\ \exists$ and $=_\alpha\ \ \mathsf{appr}_\alpha\ =$ and $\overset{\alpha}{\wedge}\ \mathsf{appr}_\alpha\ \wedge$, and since we have shown that $\mathbf{L}_\alpha^{\mathrm{sem}}\ \mathsf{appr}_\alpha\ \mathbf{L}^{\mathrm{sem}}$, together with the fact that bodies are finite, Theorem 3.8 tells us that this must hold.

Finally we show that $\mathbf{P}_\alpha^{\mathrm{sem}}\,\mathsf{appr}_\alpha\,\mathbf{P}^{\mathrm{sem}}$. Choose an arbitrary program P and atom A; we must show that

$$\mathbf{P}_\alpha^{\mathrm{sem}}\,P\,A\ \ \mathsf{appr}_\alpha\ \ \mathbf{P}^{\mathrm{sem}}\,P\,A.$$

This will hold when

$$\mathsf{lfp}\ \left(\bigsqcup_{C\in P} \mathbf{C}_\alpha^{\mathrm{sem}}\ C \right) A\ \ \mathsf{appr}_\alpha\ \ \mathsf{lfp}\ \left(\bigsqcup_{C\in P} \mathbf{C}^{\mathrm{sem}}\ C \right) A.$$

Since by Lemma 3.9 appr_α is admissible for fixed point induction, and since $\mathbf{C}_\alpha^{\mathrm{sem}}\,\mathsf{appr}_\alpha\,\mathbf{C}^{\mathrm{sem}}$, this must hold. □

4 Related Work

The earliest formal semantics for logic programs was the $\mathcal{M}$ semantics of van Emden and Kowalski [7]. This semantics expresses the denotation as the set of ground atoms entailed by the program. The $\mathcal{S}$ semantics of Falaschi *et al.* [8] is a non-ground variation on the $\mathcal{M}$ semantics and so, unlike $\mathcal{M}$ semantics, is suitable where groundness of solutions is of interest. Marriott and Søndergaard [9] propose using a set of existentially-quantified conjunctions of equations, which is nicely generalized by García de la Banda *et al.* [10] to a set of existentially-quantified conjunctions of primitive constraints, without further restricting what may serve as a primitive constraint, and without specifying how primitive constraints are to be interpreted. This is the approach we have adopted.

Probably the earliest work on analysis of logic programs was done by Warren [11] in the context of the first Prolog compiler; however, the analyses introduced there were strictly local to a single clause. The first global static analysis system, introduced by Mellish [12], was designed to infer mode declarations for Prolog predicates, as well as finding sharing among program variables. At about the same time, Søndergaard [13] applied abstract interpretation to find unifications in a program which could safely be performed without an occur-check. This analysis captured groundness, sharing, and linearity information about program variables.

The first to suggest an abstract interpretation *framework* for logic programming — the first to abstract the analysis domain from the analysis mechanism — were Jones and Søndergaard [14], extended and refined by Marriott, Søndergaard, and Jones [9].

Bruynooghe [15] proposes a rather different approach, based on an operational semantics. Bruynooghe conceives of a concrete computation as building an AND-OR tree. To avoid constructing infinite AND-OR trees, finite cyclic graphs, closely related to rational trees, are used to approximate infinite AND-OR trees. Nilsson [16] replaces the use of AND-OR trees with *context vectors*, which associate a set of possible substitutions with each point in the program. This neatly avoids any difficulties with infinite AND-OR trees.

Le Charlier and Van Hentenryck [17] present another abstract interpretation framework for logic programs, which they call GAIA. This is a top-down goal-dependent analyzer which uses tabling to avoid recomputation. García de la Banda and Hermenegildo [18] present a similar framework, called PLAI, which is more general in that it is designed to handle constraints other than just equality on Herbrand terms.

Gallagher *et al.* [19] present a goal-independent analysis framework based on a declarative semantics. This technique is based upon a *pre-interpretation* of the program, that is a mapping from the function symbols of the program to a (possibly different) domain. The domain to which they map the function symbols of the program fills the role of the abstract domain, and the pre-interpretation mapping serves as a representation function.

5 Conclusions

We have formally defined the concept of a pure constraint logic programming system and given a denotation semantics which is independent of the choice of constraint domain. Based closely on this semantics, we have presented an abstract interpretation framework. Most significantly, we have shown that when the abstract domain's abstract conjunction and projection functions precisely approximate conjunction and existential quantification in the concrete domain, the result provided by this abstract interpretation framework will be maximally precise. That is, no more precise abstraction will faithfully approximate the actual behavior of the program.

For more detail on this work, and for a discussion of goal-dependent analysis using a similar framework, see Schachte [2].

References

[1] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints, in:

Conference Record of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, CA, 1977, pp. 238–252.

[2] P. Schachte, Precise and efficient static analysis of logic programs, Ph.D. thesis, Dept. of Computer Science, The University of Melbourne, Australia (1999).

[3] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press, 1987, pp. 111–119.

[4] W. v. O. Quine, Mathematical Logic, 2nd Edition, Harvard University Press, Cambridge, 1974.

[5] R. Giacobazzi, S. Debray, G. Levi, Generalized semantics and abstract interpretation for constraint logic programs, Journal of Logic Programming 25 (3) (1995) 191–247.

[6] F. Nielson, Two-level semantics and abstract interpretation, Theoretical Computer Science 69 (2) (1989) 117–242.

[7] M. van Emden, R. Kowalski, The semantics of predicate logic as a programming language, Journal of the ACM 23 (4) (1976) 733–742.

[8] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, A new declarative semantics for logic langauges, in: R. A. Kowalski, K. A. Bowen (Eds.), Proceedings of the Fifth International Conference and Symposium on Logic Programming, ALP, IEEE, The MIT Press, Seattle, 1988, pp. 993–1005.

[9] K. Marriott, H. Søndergaard, N. Jones, Denotational abstract interpretation of logic programs, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 607–648.

[10] M. García de la Banda, K. Marriott, P. Stuckey, H. Søndergaard, Differential methods in logic program analysis, Journal of Logic Programming 35 (1) (1998) 1–37.

[11] D. H. D. Warren, Implementing Prolog — compiling predicate logic programs, D.A.I. Research Report 39, 40, University of Edinburgh (1977).

[12] C. Mellish, Some global optimizations for a Prolog compiler, Journal of Logic Programming 2 (1) (1985) 43–66.

[13] H. Søndergaard, An application of abstract interpretation of logic programs: Occur check reduction, in: B. Robinet, R. Wilhelm (Eds.), Proceedings of ESOP 86, Vol. 213 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1986, pp. 327–338.

[14] N. Jones, H. Søndergaard, A semantics-based framework for the abstract interpretation of PROLOG, in: S. Abramsky, C. Hankin (Eds.), Abstract Interpretation of Declarative Languages, Computers and Their Applications, Ellis Horwood, 1987, Ch. 6, pp. 123–142.

[15] M. Bruynooghe, A practical framework for the abstract interpretation of logic programs, Journal of Logic Programming 10 (1–4) (1991) 91–124.

[16] U. Nilsson, Towards a framework for the abstract interpretation of logic programs, in: P. D. *et al.* (Ed.), Programming Language Implementation and Logic Programming, no. 348 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1988, pp. 68–82.

[17] B. Le Charlier, P. Van Hentenryck, Experimental evaluation of a generic abstract interpretation algorithm for PROLOG, ACM Transactions on Programming Languages and Systems 16 (1) (1994) 35–101. URL `http://www.acm.org/pubs/toc/Abstracts/0164-0925/174627.html`

[18] M. García de la Banda, M. Hermenegildo, A practical approach to the global analysis of CLP programs, in: D. Miller (Ed.), Logic Programming: Proceedings of the 1993 International Symposium, MIT Press, Vancouver, Canada, 1993, pp. 437–455.

[19] J. Gallagher, D. Boulanger, H. Sağlam, Practical model-based static analysis for definite logic programs, in: Logic Programming: Proceedings of the 1995 International Symposium, The MIT Press, Portland, Oregon, USA, 1995, pp. 351–368.

A Proofs of Theorems

Proof of Theorem 2.2 :

Proof.

(i) This follows immediately from our definition of $\wedge : \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$.

(ii) Required by Axiom 2.1 (i) and (ii).

(iii) This is a basic property of set union.

(iv) This is a basic property of set union.

(v) This follows from our definition of $\wedge$.

(vi) This is basic property of set union.

(vii) This follows immediately from our definition of $\exists\ : \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$.

(viii) Choose a sequence of variables $\boldsymbol{V}$ containing all and only the variables of V, without repetition, and choose a sequence of variables $\boldsymbol{V}'$ such that $\boldsymbol{V}'$ is disjoint with V and the variable of C and C', and $\boldsymbol{V}'$ contains no repeats. By Axiom 2.1 (v), we know that $\exists \boldsymbol{V}' : C' [\boldsymbol{V}'/\boldsymbol{V}] \equiv \exists V : C'$, and further, we know that $V \cap \mathsf{vars}(C'\,[\boldsymbol{V}/V]) = \varnothing$. Therefore,

$$\begin{aligned}
\exists V : (C \wedge \exists V : C') &\equiv \exists V : (C \wedge \exists \boldsymbol{V}' : C'\,[\boldsymbol{V}'/\boldsymbol{V}]) \\
&\equiv (\exists V : C) \wedge (\exists \boldsymbol{V}' : C'\,[\boldsymbol{V}'/\boldsymbol{V}]) \\
&\equiv (\exists V : C) \wedge (\exists V : C'\,[\boldsymbol{V}'/\boldsymbol{V}]\,[\boldsymbol{V}/\boldsymbol{V}']) \\
&\equiv (\exists V : C) \wedge (\exists V : C') \\
&\equiv (\exists \boldsymbol{V}' : C\,[\boldsymbol{V}'/\boldsymbol{V}]) \wedge (\exists V : C') \\
&\equiv \exists V : ((\exists \boldsymbol{V}' : C\,[\boldsymbol{V}'/\boldsymbol{V}]) \wedge C') \\
&\equiv \exists V : ((\exists V : C\,[\boldsymbol{V}'/\boldsymbol{V}]\,[\boldsymbol{V}/\boldsymbol{V}']) \wedge C') \\
&\equiv \exists V : ((\exists V : C) \wedge C')
\end{aligned}$$

(ix) This follows from our definition of $\exists$.

(x) This follows from our definition of $\exists$.

(xi) This follows from Axiom 2.1 (vi).

(xii) This follows from Axiom 2.1 (vi).

(xiii) This follows from Axiom 2.1 (vii).

(xiv) This follows from Axiom 2.1 (viii). □

Notice that we have not used Axiom 2.1 (i) in this proof. It is in fact not necessary that $\wedge$ be commutative or absorptive; we require them because they are properties we naturally expect of a logic programming system.

Proof of Theorem 3.6 :

Proof. We must show that

(i) $\forall C \subseteq \mathsf{Con} : C \subseteq \gamma\ (\alpha\ C)$; and

(ii) $\forall A \in \mathsf{ACon} : \alpha\ (\gamma\ A) \sqsubseteq A$.

We prove these points in turn.

(i) Expanding the definitions of α and γ, we must show:

$$\forall C \subseteq \mathsf{Con} : C \subseteq \{c \in \mathsf{Con} \mid \beta\ c \sqsubseteq \bigsqcup_{c' \in C} \beta\ c'\}.$$

Choose an arbitrary $C \subseteq \mathsf{Con}$. We must show that

$$\forall c \in C : \beta\ c \sqsubseteq \bigsqcup_{c' \in C} \beta\ c'.$$

But this is an inherent property of least upper bounds.

(ii) Expanding the definitions of α and γ, we must show:

$$\forall A \in \mathsf{ACon} : \left(\bigsqcup \{\beta\ c \mid c \in \mathsf{Con} \wedge \beta\ c \sqsubseteq A\} \right) \sqsubseteq A.$$

Choose an arbitrary $A \in \mathsf{ACon}$ and consider the set on the left side of the inequality. This is a set all of whose elements are $\sqsubseteq A$. Clearly the least upper bound of this set must also be $\sqsubseteq A$. □

Proof of Theorem 3.8 :

Proof.

$$\begin{aligned}
\alpha\ (C \wedge C') &= \alpha\ \{c \wedge c' \mid c \in C \wedge c' \in C'\} && \text{(defn. of cross conjunction)}\\
&= \bigsqcup\{\beta\ (c \wedge c') \mid c \in C \wedge c' \in C'\} && \text{(definition of } \alpha)\\
&= \bigsqcup\{(\beta\ c) \curlywedge (\beta\ c') \mid c \in C \wedge c' \in C'\} && (\curlywedge \text{ is precise})\\
&= \left(\bigsqcup_{c \in C} \beta\ c\right) \curlywedge \left(\bigsqcup c' \in C' \beta\ c'\right) && (\curlywedge \text{ distributes over joins})\\
&= (\alpha\ C) \curlywedge (\alpha\ C') && \text{(definition of } \alpha)
\end{aligned}$$

□

Proof of Lemma 3.9 :

Proof. Take C to be an arbitrary chain in $\mathcal{P}(\mathsf{Con}) \times \mathsf{ACon}$ such that $\forall\, \langle a, s\rangle \in C : a\mathsf{appr}_\alpha s$, and take $\langle a_0, s_0\rangle = \bigsqcup C$. We must show that $a_0\mathsf{appr}_\alpha s_0$. Naturally, $a_0 = \bigsqcup\{a \mid \langle s, a\rangle \in C\}$ and $s_0 = \bigsqcup\{s \mid \langle s, a\rangle \in C\}$, so we need only show that $\alpha\, s_0 = a_0$. This follows from the fact that α is a lower adjoint, shown in Theorem 3.6 , and is therefore continuous. □