Precise Goal-Independent Abstract Interpretation of Constraint Logic Programs

Peter Schachte

The University of Melbourne, Victoria 3010, Australia, `schachte@cs.mu.oz.au`

Abstract

We present a goal-independent abstract interpretation framework for 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 constraint logic programming systems, give a formal semantics that is independent of the actual constraint domain and the details of the proof algorithm, and formally define the maximally precise abstraction of a constraint logic program.

1 Introduction

Optimizing compilation of computer programs requires extensive detailed information about the program to be compiled. Such information is often too tedious, complex, and error prone for programmers to write and maintain, so this task falls to program analyzers. This is especially true for very high level languages, including logic programming languages, as they allow programmers to omit many details that are unimportant to the correctness of the program.

Some of the information compilers need, such as how many times a particular loop is typically executed, or how large an array will typically be, can only be determined by observing how the program behaves in the presence of actual data. More often, however, compilers need absolute information about how a program will behave, regardless of the data provided, such as whether or not it is possible that a particular array index will ever fall outside the array. *Static analysis* is the process of determining and proving properties of a program in the absence of sample data. The most popular approach to static analysis of logic programs, and the approach we take, is *Abstract interpretation*, which we will discuss in section Section 2.

Most of the properties we wish determine, however, are undecidable in general, although in many cases they can be determined with certainty. Therefore

Preprint submitted to Elsevier Preprint *2 April 2002*

we must be satisfied with an *approximation* of a program property, and must decide on which side of that property we should err, when we cannot get an exact result. For example, if an optimizing compiler wishes to omit bounds checks on array accesses, it is acceptable, though undesirable, for an analyzer to say that an index may be out of bounds when in fact it cannot, but unacceptable to promise that it cannot be out of bounds when in fact it can. We say an analysis is *accurate* if it can never erroneously promise a desired property, and describe the extent to which it can determine properties that do hold as its *precision*.

To a great extent, the process of analyzing a program is independent of the property to be inferred, particularly when using an abstract interpretation approach. It therefore makes sense to distinguish the abstract interpretation mechanism — the *framework* — from the sort of information we seek to determine — the analysis *domain*. The framework and domain are jointly responsible for the accuracy and precision of an analysis. However, since a framework is designed infrequently, and domains are developed more often, it is desirable to prove accuracy and precision of the framework with respect to the choice of domain.

Abstract interpretation frameworks for logic programming fall into two broad categories: *goal independent* analyses determine how each predicate in the program will behave as a function of how it is called, while *goal dependent* analyses determine, given the initial goal of the program, how each individual goal in the program will be invoked. Most program optimizations tend to depend on the kind of information inferred by a goal dependent analysis. However, a goal independent analysis tends to produce results that are more understandable to the programmer, since the goal independent analysis of a predicate depends only on the predicates it invokes, while a goal dependent analysis depends on the behavior of the entire program.

A third approach, called *condensing* [1], combines the other two. Since a goal independent analysis determines how a predicate will behave in each context, this information could be used in performing a goal dependent analysis. This can speed up the goal dependent analysis, even after accounting for the time spent on the separate goal independent analysis phase. This is especially true in a practical context: since a goal independent analysis depends only on the predicates it calls (directly and indirectly), the goal independent analysis of unchanging code, such as library code, need never be repeated. Unfortunately, the condensing approach is particularly sensitive to imprecision in the analysis domain.

The aim of this paper, then, is to present a framework for goal-independant abstract interpretation of (constraint) logic programs in general, and to provide a set of sufficient conditions on a domain to ensure that this framework will always produce accurate and maximally precise results.

The remainder of this paper is arranged as follows. Section 2 provides an introduction to abstract interpretation. Section 3 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 4 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. Finally, Section 5 describes related work, and Section 6 presents our conclusions.

2 Abstract Interpretation

Abstract interpretation [2] 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 [1] 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 a complete lack of 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 cases of unreachable, failing, (infinitely) looping, or error-causing code. This value is usually denoted $\bot$.

Another way to think about this is that we are partitioning the set of all

[1] When there are an infinite number of abstract values, termination may not be guaranteed. Even when there are finitely many abstractions, there may be so many that direct computation is not tractable. In these cases we may guarantee termination in reasonable time through *widening* or *narrowing*. This topic is outside the scope of this paper, but see [2,3] for a discussion.

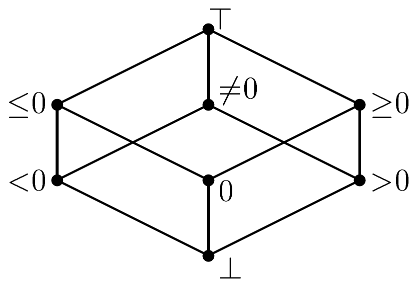

Fig. 1. Hasse diagram for the signs abstract domain for a single variable

integers $\mathbb{Z}$ 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 $\{\bot, <0, =0, >0, \leq 0, \neq 0, \geq 0, \top\}$.

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, $\mathbb{Z}$ in this example; Abst refers to the abstract domain, in this case $\{\bot, <0, =0, >0, \leq 0, \neq 0, \geq 0, \top\}$.

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 an abstraction at all!). We will also need a function to abstract the concrete values appearing in the program to be analyzed. We define an *abstraction function* as

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

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

Fig. 2. Correctness condition for function approximation

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, for the purposes of this paper, 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. [2] 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

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

[2] The $\sqsubseteq$ symbol here compares the precision of two abstractions. Saying $a \sqsubseteq b$ means that a is at least as strong an abstraction as b — it abstracts no concrete values that b does not.

To further pursue our example of signs, the γ function for our abstract domain would be defined as:

$$\begin{aligned}
\gamma(\top) &= \mathbb{Z} \\
\gamma(\neq 0) &= \mathbb{Z} \setminus \{0\} \\
\gamma(\leq 0) &= \{n \mid n \leq 0\} \\
\gamma(\geq 0) &= \{n \mid n \geq 0\} \\
\gamma(<0) &= \{n \mid n < 0\} \\
\gamma(>0) &= \{n \mid n > 0\} \\
\gamma(=0) &= \{0\} \\
\gamma(\bot) &= \varnothing
\end{aligned}$$

and α would be defined as:

$$\alpha(S) = \begin{cases}
\bot & \textsf{when } S = \varnothing \\
=0 & \textsf{when } S = \{0\} \\
<0 & \textsf{when } S \neq \varnothing \wedge \forall s \in S : s < 0 \\
>0 & \textsf{when } S \neq \varnothing \wedge \forall s \in S : s > 0 \\
\leq 0 & \textsf{when } S \neq \varnothing \wedge \forall s \in S : s \leq 0 \\
\geq 0 & \textsf{when } S \neq \varnothing \wedge \forall s \in S : s \geq 0 \\
\neq 0 & \textsf{when } S \neq \varnothing \wedge \forall s \in S : s \neq 0 \\
\top & \textsf{otherwise}\,.
\end{cases}$$

Notice that this abstract domain, as presented, can only describe the state of a single variable. Since we typically have many variables in a program, we need an abstraction of the states of many variables at once. In the concrete domain, it is convenient to represent the state of a program as a *valuation function* which maps variables to their values. An *abstract valuation* is then an abstraction of a valuation function. One obvious way to abstract a valuation function given an abstract domain for individual variables is as a function mapping variables to abstract values, ordered pointwise. This is not the only form of abstract valuation, however.

It is worth noting that while the γ function described above is additive, not all γ functions are. For example, if we consider the two abstract values <0 and $=0$, the union of the sets of concrete values described by these abstract values is exactly the same as the set described by the join of these abstractions, which is ≤ 0. However, when we extend this domain to handle multiple variables as just discussed, this will no longer hold. For example, suppose we abstract a pair of values as a pair of abstractions ordered componentwise. The least upper bound of the abstractions $\langle =0, >0\rangle$ and $\langle <0, =0\rangle$ is $\langle \leq 0, \geq 0\rangle$. Notice that this abstraction describes the concrete values $\langle -1, 1\rangle$, while neither $\langle =0, >0\rangle$ nor $\langle <0, =0\rangle$ does, thus this concretization function is not additive.

3 The Semantics of Constraint Logic Programs

We will begin with an overview of the notation we will in the sequel.

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 [4]. 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. 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.

One might wonder why we choose to define Body as a set of *sets* rather than a set of *sequences* of literals. A sequence of literals will only handle a simple left-to-right selection rule as used for Prolog. In order to handle more sophisticated selection rules, such as is used by Mercury [5], we would need to be supplied with an explicit selection function which is responsible for determining the order of execution of literals in a body. For the purposes of goal independent analysis, we can ignore this by adding the requirements of Axiom 1 (1) below.

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.

The goal of this paper is to provide a denotational semantics and corresponding abstract interpretation framework for constraint logic programming in general; that is, the semantics should be as independent as possible of the implementation details of the language. This includes the language's selection and search rules, which determine in what order the parts of the program are executed, as well as the choice of constraint domain, which determines the primitive predicates of the language. However, there are several properties which we require of any constraint logic programming system we are to analyze. Below we shall define the semantics of an atom as just the set of alternative constraints it creates as it executes. Therefore, we can ensure that a logic programming system exhibits certain properties by requiring that it treat certain classes of programs equivalently, *i.e.*, by requiring the following equivalences on Con:

Axiom 1 (Logic programming language) *For* $c, c', c'' \in \mathsf{Con}$, $t, t' \in \mathsf{Term}$, *and* $v, v' \in \mathsf{Var}$*:*

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

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

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

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

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

□

Axiom 1 (1) is not entirely necessary for the semantics and analysis framework we are developing. Requiring conjunction to be commutative and absorptive allows us to view a clause body as a set of literals, and saves us from worry about a selection rule. These requirements could be removed at the cost of

[3] We indicate elements of a program being analyzed, as distinguished from the elements of the analysis framework being presented, using *Quine corners* [6], written $\ulcorner\cdot\urcorner$. Quine corners are not quotes; the enclosed material may include variables intended to be interpreted as meta-variables.

some added complication in the semantics and analyzer. This would be necessary before this approach could be applied to Prolog, for which conjunction is neither commutative nor absorptive. The necessary modification is straightforward and will not be discussed.

However, conjunction must be associative for our approach to goal independent analysis to work. Fortunately, conjunction is associative in all logic programming systems we know.

The rest of Axiom 1 should be uncontroversial and easily satisfied by existing logic programming languages. For example, Axiom 1 (2) ensures that `true` and other trivially satisfied goals cannot change the meaning of a program, and that failure halts forward progress on the failed alternative. We could also require false to be a right annihilator, but this would cause difficulty if we later wish to accommodate languages with side effects, as constructs such as

```
member(X, L), write(X), fail.
```

would be equivalent to false. Axiom 1 (2) ensures projecting away a variable that does not exist does not change anything. Axiom 1 (4) ensures that two variables to be projected away can be removed in either order, allowing us to project away a set of variables all at once. Axiom 1 (5) effectively requires that a quantified variable can be uniformly renamed throughout its scope without changing the meaning of the code.

However, Prolog's cut (`!`) primitive, negation, and if-then-else facility are not so easily handled in this framework. Similarly, predicates that have side effects, such as performing input/output or changing the running program, are difficult to handle in this framework. This is left as future work.

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 $S \subseteq \mathsf{Con}, v \in \mathsf{Var}$, 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 $t\left[\vec{t}/\vec{v}\right]$ the pairwise substitution of terms from $\vec{t}$ for variables from $\vec{v}$ in t, where $\vec{t}$ is a sequence of terms, $\vec{v}$ is a sequence of variables of the same length, no variable appears more than once in $\vec{v}$, and $\vec{v}$ and $\mathsf{vars}(\vec{t})$ 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 $\vec{s}$ such that $\vec{s}$ 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, $t\,[\vec{v}'/\vec{v}]\,[\vec{v}/\vec{v}'] = t$ whenever $\vec{v}$ and $\vec{v}'$ are disjoint, and $\vec{v}'$ and $\mathsf{vars}(t)$ are disjoint.

We require one further function:

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

produces a variant of the input clause that has no variables in common with the given set of variables. We use this function later to enforce the rule that the scope of a variable is limited to the clause it occurs in.

The semantics of a program can be thought of as a function mapping an atom to the set of all constraints that satisfy that atom, so we specify our semantic domain **Den** as:

$$\mathbf{Den} = \mathsf{Atom} \to \mathcal{P}(\mathsf{Con}).$$

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

Definition 2 (Program semantics) *The denotation of a program is given by the function* $\mathbf{P}^{\mathrm{sem}}$*, which we define in terms of the auxiliary functions* $\mathbf{C}^{\mathrm{sem}}$ *and* $\mathbf{L}^{\mathrm{sem}}$*:*

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

These functions are defined as follows:

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

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

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

□

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).

Example 1 *Consider the venerable naive reverse predicate:*

$$nrev([], []) \leftarrow \mathsf{true}. \qquad (C_1)$$
$$nrev([u|v], y) \leftarrow nrev(v, w)\ \wedge\ append(w, [u], y). \qquad (C_2)$$

$$append([], y, y) \leftarrow \mathsf{true} \qquad (C_3)$$
$$append([u|v], y, [u|w]) \leftarrow append(v, y, w). \qquad (C_4)$$

To compute the semantics by Kleene iteration we begin by computing the least upper bound of $\mathbf{C}^{\text{sem}}\ C_i\ \bot$ *for each clause* C_i *in the program. Considering the first clause, we get:*

$$\begin{aligned}\mathbf{C}^{\text{sem}}\ C_1\ \perp\ A &= \mathbf{L}^{\text{sem}}\ \ulcorner nrev([],[]) = A\urcorner\ \perp\ \wedge\ \{\mathsf{true}\}\\ &= \{nrev([],[]) = A\}\end{aligned}$$

Turning to the second clause, we get:

$$\mathbf{C}^{\text{sem}}\ C_2\ \perp\ A = \exists uvwy : \begin{pmatrix} \quad \mathbf{L}^{\text{sem}}\ \ulcorner nrev([u|v],y) = A\urcorner\ \perp \\ \wedge\ \mathbf{L}^{\text{sem}}\ \ulcorner nrev(v,w)\urcorner\ \perp \\ \wedge\ \mathbf{L}^{\text{sem}}\ \ulcorner append(w,[u],y)\urcorner\ \perp \end{pmatrix}$$

But since $nrev(v,w) \in \mathsf{Atom}$, $\mathbf{L}^{\text{sem}}\ \ulcorner nrev(v,w)\urcorner\ \perp = \perp\ nrev(v,w) = \varnothing$. *Therefore,*

$$\mathbf{C}^{\text{sem}}\ C_2\ \perp\ A = \varnothing.$$

For the third clause, we find that

$$\mathbf{C}^{\text{sem}}\ C_3\ \perp\ A = \{\exists y : append([],y,y) = A\}$$

and for the final clause, we again get

$$\mathbf{C}^{\text{sem}}\ C_4\ \perp\ A = \varnothing.$$

Computing the least upper bound of these functions, we get:

$$\bigsqcup_{C \in P} \mathbf{C}^{\text{sem}}\ C\ \perp\ A = \{nrev([],[]) = A, \exists y : append([],y,y) = A\}.$$

Using this result as the new d *function, call it* d_1*, we evaluate all this again. Since the evaluation of* $\mathbf{C}^{\text{sem}}\ C_1\ d$ *and* $\mathbf{C}^{\text{sem}}\ C_3\ d$ *did not refer to* d *at all, they yield the same result as above. For clause* C_2*, we now get*

$$
\begin{aligned}
\mathbf{C}^{\text{sem}}\ C_2\ d_1\ A &= \exists uvwy : \left(\begin{array}{l} \quad \mathbf{L}^{\text{sem}}\ \ulcorner nrev([u|v],y) = A\urcorner\ d_1 \\ \wedge\ \mathbf{L}^{\text{sem}}\ \ulcorner nrev(v,w)\urcorner\ d_1 \\ \wedge\ \mathbf{L}^{\text{sem}}\ \ulcorner append(w,[u],y)\urcorner\ d_1 \end{array}\right) \\
&= \exists uvwy : \left(\begin{array}{l} \quad \{nrev([u|v],y) = A\} \\ \wedge\ d_1\ nrev(v,w) \\ \wedge\ d_1\ append(w,[u],y) \end{array}\right) \\
&= \exists uvwy : \left(\begin{array}{l} \quad \{nrev([u|v],y) = A\} \\ \wedge \left\{\begin{array}{l} \exists y' : append([],y',y') = nrev(v,w), \\ nrev([],[]) = nrev(v,w) \end{array}\right\} \\ \wedge \left\{\begin{array}{l} \exists y' : append([],y',y') = append(w,[u],y), \\ nrev([],[]) = append(w,[u],y)) \end{array}\right\} \end{array}\right) \\
&= \exists uvwy : \left(\begin{array}{l} \quad \{nrev([u|v],y) = A\} \\ \wedge\ \{\mathsf{false}, v = [] \wedge w = []\} \\ \wedge\ \{w = [] \wedge y = [u], \mathsf{false}\}) \end{array}\right) \\
&= \exists uvwy : \{nrev([u|v],y) = A \wedge v = [] \wedge w = [] \wedge w = [] \wedge y = [u]\} \\
&= \{\exists u : nrev([u],[u]) = A\}
\end{aligned}
$$

For the fourth clause, we get:

$$
\begin{aligned}
\mathbf{C}^{\text{sem}}\ C_4\ d_1\ A &= \exists uvwy : \left(\begin{array}{l} \quad \mathbf{L}^{\text{sem}}\ \ulcorner append([u|v],y,[u|w]) = A\urcorner\ d_1 \\ \wedge\ \mathbf{L}^{\text{sem}}\ \ulcorner append(v,y,w)\urcorner\ d_1 \end{array}\right) \\
A &= \exists uvwy : \left(\begin{array}{l} \quad \{append([u|v],y,[u|w]) = A\} \\ \wedge \left\{\begin{array}{l} \exists y' : append([],'y,y') = append(v,y,w), \\ nrev([],[]) = append(v,y,w) \end{array}\right\} \end{array}\right) \\
&= \{\exists uw : append([u],w,[u|w]) = A\}
\end{aligned}
$$

Computing the least upper bound of these functions, we get:

$$\bigsqcup_{C \in P} \mathbf{C}^{\text{sem}}\ C\ d_1\ A = \left\{ \begin{array}{l} nrev([],[]) = A, \\ \exists u : nrev([u],[u]) = A, \\ \exists y : append([], y, y) = A, \\ \exists uw : append([u], w, [u|w]) = A \end{array} \right\}$$

This process continues ad infinitum *generating ever larger sets with ever larger terms.* $\square$

4 Abstract Interpretation of Constraint Logic Programs

In abstract interpretation, we wish to find an abstract version of the semantics of a program. 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 (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* $F : S \to \mathsf{ACon}$ *and* $G : S \to \mathcal{P}(\mathsf{Con})$ *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 $F : \mathsf{ACon} \to \mathsf{ACon}$ *and* $G : \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$ *as follows:*

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

$\square$

Now we specify what we require of an abstraction.

Definition 4 (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* $\curlywedge : \mathsf{ACon} \to \mathsf{ACon} \to \mathsf{ACon}$.
- *a* projection function $\mathsf{project} : \mathcal{P}(\mathsf{Var}) \to \mathsf{ACon} \to \mathsf{ACon}$. *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 [7] 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 5 *We define a* representation function $\beta : \mathsf{Con} \to \mathsf{ACon}$ *which gives a maximally precise*[5] *abstraction for each concrete conjunction of constraints as follows:*

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

□

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

Definition 6 *From the representation function* β*, we define the needed concretization function* $\gamma : \mathsf{ACon} \to \mathcal{P}(\mathsf{Con})$ *and abstraction function* $\alpha : \mathcal{P}(\mathsf{Con}) \to \mathsf{ACon}$ *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 7 α *and* γ *form a Galois connection, with* α *the lower adjoint and* γ *the upper.*

[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 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.

PROOF. We must show that

(1) $\forall C \subseteq \mathsf{Con} : C \subseteq \gamma\ (\alpha\ C)$; and
(2) $\forall A \in \mathsf{ACon} : \alpha\ (\gamma\ A) \sqsubseteq A$.

We prove these points in turn.

(1) 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.

(2) 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 $\sqsubseteq A$. □

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

Definition 8 (precise abstraction) *A* precise abstraction *is an abstraction for which equivalence on* Acon *precisely approximates equivalence on* Con*. That is, if two constraints are equivalent, then so must their abstractions be.* □

The intuition behind this definition is that if two equivalent constraints have different abstractions, at least one of them must not be the most precise abstraction for that constraint. Conversely, if there were a more precise abstraction for a given constraint, it would be possible to produce it from an equivalent constraint specifying extra redundant constraints.

For example, consider the analysis domain that approximates a unification by the set of variables that it grounds. For this analysis domain, abstract conjunction would naturally be union. However, this domain is not precise, as this Prolog code demonstrates:

```
X = Y, X = a.
```

Abstracting the first unification gives us $\varnothing$ as the set of ground variables. The second unification gives us $\{X\}$, as does the abstract conjunction of both goals. However, the given constraint is equivalent to

```
X = Y, X = a, Y = a.
```

which has the abstraction $\varnothing \cup \{X\} \cup \{Y\} = \{X, Y\}$. This is the most precise approximation for these equivalent constraints possible in this domain. The fact that two equivalent constraints have different abstractions implies that this domain is not precise.

This definition of precise abstraction has the following consequences:

Theorem 9

$\hat{\wedge}$ distributes over finite and infinite joins, that is, for all (possibly infinite) $A, A' \subseteq \mathsf{ACon}$,

$$\left(\bigsqcup A\right) \hat{\wedge} \left(\bigsqcup A'\right) = \bigsqcup\{a \hat{\wedge} a' \mid a \in A \wedge a' \in A'\}.$$

PROOF. Firstly note that due to our definition of $\wedge : \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$, we have that $\wedge$ distributes over possibly infinite unions. But since equivalence on Acon precisely approximates equivalence on Con, $\hat{\wedge}$ must similarly distribute over finite and infinite joins. $\square$

Theorem 10 *The abstract conjunction function precisely approximate conjunction. That is,*

$$\forall C, C' \subseteq \mathsf{Con} : \alpha\ (C \wedge C') = (\alpha\ C) \hat{\wedge} (\alpha\ C')$$

must hold.

PROOF.

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

$\square$

Theorem 11

$\mathsf{project}$ *distributes over finite and infinite joins, that is, for all (possibly infinite)* $A \subseteq \mathsf{ACon}$*,*

$$\bigsqcup\left(\mathsf{project}\, V A\right) = \mathsf{project}\, V\left(\bigsqcup A\right)$$

PROOF. Firstly note that due to our definition of $\exists : \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Con}) \to \mathcal{P}(\mathsf{Con})$, we have that $\exists$ distributes over possibly infinite unions. But since equivalence on Acon precisely approximates equivalence on Con, $\overset{\wedge}{\alpha}$ must similarly distribute over finite and infinite joins. □

Theorem 12 *The abstract projection function precisely approximates existential quantification (*$\mathsf{project}\ \mathsf{appr}_\alpha\ \exists$*). That is,*

$$\forall V \subseteq \mathsf{Var}, C \subseteq \mathsf{Con} : \mathsf{project}\ \ V\ \ (\alpha\ C) = \alpha\ (\exists V : C).$$

PROOF.

$\alpha\ (\exists V : C)$

$$
\begin{aligned}
&= \bigsqcup\{\beta\ (\exists V : c) \mid c \in C\} && \text{(definition of } \alpha) \\
&= \bigsqcup\{\mathsf{project}\, V(\beta\ c) \mid c \in C\} && \text{(definition of } \beta) \\
&= \mathsf{project}\, V \bigsqcup_{c\in C} \beta\ c && (\mathsf{project} \text{ distributes over joins}) \\
&= \mathsf{project}\, V(\alpha\ C) && \text{(definition of } \alpha)
\end{aligned}
$$

□

Now we define an abstract semantics to be a function

$$\textbf{ADen} = \mathsf{Atom} \to \mathsf{ACon}.$$

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 8. 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 13 (abstract semantic function) *We specify the* abstract semantics *of a program as the result of the (goal independent) abstract semantic function* $\mathbf{P}^{\mathrm{sem}}_\alpha$*, which we define in terms of the auxiliary functions* $\mathbf{C}^{\mathrm{sem}}_\alpha$ *and* $\mathbf{L}^{\mathrm{sem}}_\alpha$*:*

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

These functions are defined as follows:

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

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

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

□

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

Lemma 14 *The* appr_{α} *relation on* $\mathsf{ACon} \times \mathcal{P}(\mathsf{Con})$, *ordered componentwise, is admissible for fixed point induction.*

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 7, and is therefore continuous. □

Now we are equipped to prove the main result of this paper: that the abstract semantics given in Definition 13, when applied to any abstraction satisfying Definition 8, will always yield the most precise abstraction of any given program.

Theorem 15 $\mathbf{L}^{\mathrm{sem}}_{\alpha}\ \mathsf{appr}_{\alpha}\ \mathbf{L}^{\mathrm{sem}}$, $\mathbf{C}^{\mathrm{sem}}_{\alpha}\ \mathsf{appr}_{\alpha}\ \mathbf{C}^{\mathrm{sem}}$, *and* $\mathbf{P}^{\mathrm{sem}}_{\alpha}\ \mathsf{appr}_{\alpha}\ \mathbf{P}^{\mathrm{sem}}$.

PROOF. First we prove $\mathbf{L}^{\mathrm{sem}}_{\alpha}\, \mathsf{appr}_{\alpha}\, \mathbf{L}^{\mathrm{sem}}$. Choose an arbitrary $L \in \mathsf{Lit}$ and $d \in \textbf{Den}$, 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 $\mathbf{L}^{\text{sem}}_\alpha\,\mathsf{appr}_\alpha\,\mathbf{L}^{\text{sem}}$, we also have $\mathbf{C}^{\text{sem}}_\alpha\,\mathsf{appr}_\alpha\,\mathbf{C}^{\text{sem}}$. Choose an arbitrary $C \in \mathsf{Clause}$, $d \in \textbf{Den}$, 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

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

By Theorems 12 and 10 we have that $\mathsf{project}\ \ \mathsf{appr}_\alpha\ \exists$ and $\mathbin{\overset{\alpha}{\wedge}}\ \mathsf{appr}_\alpha\ \wedge$. Thus since by the definitions of α and β we have that $H{=}_\alpha A\ \mathsf{appr}_\alpha\ H{=}A$, and since we have shown that $\mathbf{L}^{\text{sem}}_\alpha\ \ \mathsf{appr}_\alpha\ \mathbf{L}^{\text{sem}}$, then $\mathbf{L}^{\text{sem}}_\alpha\,\mathsf{appr}_\alpha\,\mathbf{L}^{\text{sem}}$.

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

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

This will hold when

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

Since by Lemma 14 appr_α is admissible for fixed point induction, and since $\mathbf{C}^{\text{sem}}_\alpha\,\mathsf{appr}_\alpha\,\mathbf{C}^{\text{sem}}$, this must hold. $\square$

5 Related Work

The earliest formal semantics for logic programs was the $\mathcal{M}$ semantics of van Emden and Kowalski [8]. This semantics expresses the denotation as the set of ground atoms entailed by the program. The $\mathcal{S}$ semantics of Falaschi et al. [9] 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 [10] propose using a set of existentially-quantified conjunctions of equations, which is nicely generalized by García de la Banda et al. [11] 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 [12] 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 [13], was designed to infer mode declarations for Prolog predicates, as well as finding sharing amongst program variables. At about the same time, Søndergaard [14] 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 [15], extended and refined by Marriott, Søndergaard, and Jones [10].

Bruynooghe [16] 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 [17] 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 [18] 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 [19] 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.

Winsborough [20] presents another goal-dependent analysis framework, using a *minimal function graph semantics*. Such a semantics associates with each clause a partial function mapping invocation states to success states, which is quite similar to our definition of **Den**. Winsborough's approach allows a compiler to generate multiple versions of each predicate, each specialized to a different use, with the selection being made at compile time. The problem of minimizing the number of versions of each predicate generated is solved by equating it to the well-understood problem of minimizing deterministic finite automata.

Gallagher et al. [21] 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. Thus this approach is similar to ours, albeit with considerably less machinery. In this approach, however, the program to be analyzed is translated into a program on the smaller "abstract" domain, and then the models of the program are computed. Because the number of models of a predicate may be exponential in the arity of the predicate, practical use of this approach would require a specialized representation of sets of models, which the authors have not discussed. However, they do describe a number of optimizations, such as using a nonground representation for models, which reduce this cost.

Codish and Demoen [22] show that a goal-independent analyzer can be made to perform goal-dependent analysis by transforming the program to be analyzed using magic sets. This transformation works by enriching the program to be analyzed with an extra clause for each goal in the program, plus an extra clause for each possible initial goal for the program. Each of these new clauses specifies in what cases, and in what states, a particular goal can be reached, in terms of the cases and states in which the head of the clause containing that goal can be reached and the success of all goals in that clause preceding the goal of interest.

The work most closely related to the present paper is that of Anno Langen [1]. Langen suggested that some analysis domains could be "condensed" as described in Section 1. Langen proved a set of criteria sufficient to ensure a condensing analysis would not lose precision. Langen's overall approach, however, was rather different from the one presented in this paper, and so his criteria are different. Langen's approach does not consider constraints in general, but only equality constraints. Furthermore, his abstract unification operation combines the abstract equality constraint with a meet operation, avoiding the need for a separate abstract conjunction operation. Given this overall approach, Langen proved that if an abstract unification operation is commutative, idempotent, and additive, then the domain is condensing. We are confident that a precise goal independent analysis, together with a carefully designed goal dependent analysis framework using the results of the goal independent analysis, will ensure a condensing analysis without loss of precision. A similar proof is given in [23].

6 Conclusions

We have given a denotation semantics for a broad class of constraint logic programming languages which is independent of the choice of constraint domain and selection and search rules. Based 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.

Much work remains to be done. Firstly, our analysis framework needs to be extended to handle pruning operators such as Prolog's cut (`!`), side effects, and language-specific selection rules. Also a goal dependent analysis framework matching the goal independent framework of the present paper should be developed.

Acknowledgements

The author would like to thank Harald Søndergaard for many helpful discussions of the topics covered in this paper. The several anonymous referees who have read various versions of this paper have made numerous helpful suggestions.

References

[1] A. Langen, Advanced techniques for approximating variable aliasing in logic programs, Ph.D. thesis, University of Southern California (March 1991).

[2] 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.

[3] R. Giacobazzi, Semantic aspects of logic program analysis, Ph.D. thesis, University of Pisa, Italy (1993).

[4] 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.

[5] Z. Somogyi, F. Henderson, T. Conway, The execution algorithm of Mercury: An efficient purely declarative logic programming language, Journal of Logic Programming 29 (1–3) (1996) 17–64, uRL: `http://www.cs.mu.oz.au/mercury/papers/jlp.ps.gz`.

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

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

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

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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`

[19] 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.

[20] W. Winsborough, Multiple specialization using minimal-function graph semantics, Journal of Logic Programming 13 (2–3) (1992) 103–179.

[21] 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.

[22] M. Codish, B. Demoen, Analysing logic programs using "Prop"-ositional logic programs and a magic wand, in: D. Miller (Ed.), Logic Programming: Proceedings of the 1993 International Symposium, MIT Press, 1993, pp. 114–129.

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