On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs
(Extended Abstract)

F. Bueno* P. Deransart† W. Drabent‡ G. Ferrand§
M. Hermenegildo* J. Małuszyński¶ G. Puebla*

1 Introduction

This paper presents some on-going work in the ESPRIT project DiSCiPl. The project aims at devising advanced tools for debugging of constraint logic programs.

A central problem in program development is obtaining a program which satisfies the user's expectations. When considering a given program, a natural question is then whether or not it fulfils expectations of some kind (requirements). To be able to formulate this question, some formal or informal way of specifying such requirements is needed. That is, a (formal or informal) program *semantics* is needed, in which one can express what the program computes and what it is required to compute.

It may then be possible either to *verify* that the program satisfies the requirement for every computation (in the considered class), or to show a specific computation where the requirement is violated. The process of identifying the part of the program responsible for the violation is referred to as *diagnosis*. The program then needs to be modified to correct the error. Since the requirement documentation is often not complete, the user's requirements are often given as *approximations*, i.e., safe specifications of (parts of) the intended semantics of the program. The process of *debugging* consists of the study of the program semantics, observation of error symptoms, localization of program "errors" and their correction until no symptom can be observed anymore and the program is considered correct.

Semantic approximations have been used in program validation, in declarative diagnosis, and in program analysis. This paper gives a common view of these techniques from the perspective of debugging. The objective is to explore possible uses of approximations for debugging purposes. The presentation is organized as follows. First, some notions on program semantics are given, mainly by means of examples. Then, validation, diagnosis by proofs, and declarative diagnosis are described in terms of set-theoretic relations. Next, the effect of using approximations rather than the exact sets is studied. Finally, such relations on set approximations are reformulated for the special case of abstract interpretation.

*Facultad de Informática, Universidad Politécnica de Madrid, 28660-Boadilla del Monte, Madrid, Spain. {bueno,herme,german}@fi.upm.es
†INRIA-Rocquencourt, Projet LOCO, BP 105, F-78153 Le Chesnay Cedex, France. Pierre.Deransart@inria.fr
‡Institute of Computer Science, Polish Academy of Sciences. Poland. wdr@mimuw.edu.pl
§LIFO, University of Orléans, Rue de Chartres B.P. 6759, 45067 Orleans Cedex 2, France. Gerard.Ferrand@lifo.univ-orleans.fr
¶Linköping University, Department of Computer and Information Science, Östergötland, S 581 83 Linköping, Sweden. jmz@ida.liu.se

We keep the basic discussion quite general, in that we impose only some minor restrictions on the way the different semantics are formalized. We illustrate the general discussion by very simple examples referring to Constraint Logic Programming (CLP) [JM94].

2 Actual and Intended Semantics

Semantics associate a *meaning* to a given syntax (generally of a program). A particular semantics captures some features of the computations of a program (sometimes called the "observables") while hiding others. Different kinds of semantics can be used depending of the features to be described.

In this paper we restrict ourselves to the important class of semantics referred to as *fixpoint semantics*. In this approach a (monotonic) semantic operator (which we refer to as S_P) is associated with each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Example 2.1 *An example of a set-based, fixpoint semantics for (constraint) logic programs is the traditional least model semantics [JM94]. The semantic objects in this case are so called D-atoms. A D-atom is an expression $p(d_1, \ldots, d_n)$ where p is an n-ary predicate symbol, $d_1, \ldots, d_n \in D$ and D is the domain of values. For example, in classical logic programming D is the Herbrand universe; for CLP(R) D is the set of real numbers and of terms (for example lists) containing real numbers*[1]*. The semantic domain is the lattice of sets of D-atoms.*

The semantic operator for program P is T_P (the immediate consequence operator) and $[\![P]\!] = \mathrm{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\emptyset)$. An important property is that $[\![P]\!]$ is the least D-model of the program. Any ground instance[2] *of a computed answer (for an atomic query) is a member of $[\![P]\!]$.*

For example, given the following CLP program, over the domain of integers:

$$
\begin{aligned}
&sorted(X) \leftarrow X = [\,].\\
&sorted(X) \leftarrow X = [Y].\\
&sorted(X) \leftarrow X = [H1|T1], T1 = [H2|T2], H1 > H2, sorted(T1).
\end{aligned}
$$

we have that $[\![P]\!] = \{\, sorted([\,]) \,\} \cup \{\, sorted([X]) \mid X \in D \,\} \cup \{\, sorted([X_1, \ldots, X_n]) \mid n \geq 2,\ X_1 > \cdots > X_n \,\}$. So for instance $[\![P]\!]$ contains $sorted([7])$, $sorted([a])$, $sorted([[\,]])$, $sorted([2,1,0])$ and does not contain $sorted([0,2])$, $sorted([2,1,a])$.

Example 2.2 *Another example of a fixpoint semantics is the traditional "call-answer operational semantics" for CLP programs (see, e.g., [GHB+96]). The semantic objects in this case are pairs of constrained atoms. The program is assumed to contain a query or "entry point". $[\![P]\!]$ contains all the call-answer pairs that appear during program execution for the given query or entry point. For example, given the CLP program above and the query "$\leftarrow X = [1,Y], sorted(X)$", and, assuming standard left-to-right, depth-first control, we have $[\![P]\!] = \{(sorted(X) \leftarrow X = [1,Y], sorted(X) \leftarrow X = [1,Y] \wedge Y < 1), (sorted(X) \leftarrow X = [Y], sorted(X) \leftarrow X = [Y])\}$.*

Both program validation and diagnosis, to be discussed more precisely later, compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. The nature of the requirements considered in validation and diagnosis is very wide. For example, one can discuss *declarative* diagnosis/validation (when the requirements concern the

[1]Usually it is assumed that D is given together with a fixed interpretation of the symbols that can occur in constraints. For instance for CLP(R), $+$ is interpreted as addition and $>$ as the "greater than" relation on reals.

[2]In CLP, by a ground instance of a constrained atom $A \leftarrow c$ we mean any D-atom $A\theta$ such that $c\theta$ is true; here A is an atom, c a constraint and θ is a valuation assigning elements of D to variables.

relation specified by the program), diagnosis/validation of *dynamic properties* (when the requirements concern properties of the execution states), *performance* diagnosis/validation (when the requirements concern the efficiency of execution), etc. Thus, different kinds of user's expectations require different kinds of semantics in order to be able both to adequately express the requirements and to extract relevant meaning from the program to compare with the requirements.

Example 2.3 *In CLP, requirements regarding characteristics of the computed answers of a program can in general be expressed and checked using the least D-model semantics of Example 2.1, whereas if the requirements also refer to characteristics of the calls that occur during execution then the operational semantics of Example 2.2 (using sets of pairs of constrained atoms) would need to be used.*

We focus here on the common case in which the actual semantics $[\![P]\!]$ of a program is a set (and the semantic domain is the lattice of sets ordered by set inclusion). A natural question is thus how the user's intention can be represented. For the time being, let us assume that $\mathcal{I}$ belongs to the same semantic domain used for $[\![P]\!]$. The semantic object $\mathcal{I}$ can be seen as the corresponding semantics of an intended program. But this program does not exist (neither as program, nor in mind) in general. Thus, usually there is no expression of $\mathcal{I}$, but rather partial descriptions of it.

Example 2.4 *If the program of Example 2.1 is intended to compute all integer lists that are sorted, the programmer may approximate this intention with:*

$$\mathcal{I}_1 = \{sorted([X]) \mid X \textit{ is an integer}\}$$

$$\mathcal{I}_2 = \{sorted(L) \mid L \textit{ is an integer list}\}$$

Obviously, $\mathcal{I}_1$ represents a subset of the programmer's intention, since it represents only sorted integer lists of length one. Similarly, $\mathcal{I}_2$ represents a superset of the programmer's intention; it does not require that the lists are sorted.

3 Validation and Diagnosis in a Set Theoretic Framework

This section summarizes well-known notions related to program validation (see, e.g., [Der93]), diagnosis by proof, and declarative diagnosis [Sha82, Fer87]. The problems found in these disciplines are summarized and discussed in a set theoretic framework for clarity. They can also be formulated in a lattice theoretic setting, but the set theoretic presentation simplifies the discussion.

3.1 Validation

Validation aims at proving certain properties of a program which are formally defined as relationships between a specification $\mathcal{I}$ and the actual program semantics $[\![P]\!]$. Table 1 lists validation problems in a set theoretic formulation.

Property	**Definition**
P is partially correct w.r.t. $\mathcal{I}$	$[\![P]\!] \subseteq \mathcal{I}$
P is complete w.r.t. $\mathcal{I}$	$\mathcal{I} \subseteq [\![P]\!]$
P is incorrect w.r.t. $\mathcal{I}$	$[\![P]\!] \not\subseteq \mathcal{I}$
P is incomplete w.r.t. $\mathcal{I}$	$\mathcal{I} \not\subseteq [\![P]\!]$

Table 1: Set theoretic formulation of validation problems

Note that we do not assume that $\mathcal{I}$ is unique. We simply denote specifications as $\mathcal{I}$, but it can very well be the case that different specifications are given for verifying different properties. In particular, when dealing with partial correctness, $\mathcal{I}$ describes a property which should be satisfied by all elements of the semantics $[\![P]\!]$. In other words, $\mathcal{I}$ corresponds to expected properties of all results or all behaviours of the program (depending of the kind of semantics). When dealing with completeness $\mathcal{I}$ characterizes a set of elements which should be in the semantics $[\![P]\!]$, i.e., $\mathcal{I}$ describes some expected results or behaviours of P. Proving incorrectness and incompleteness is also of interest, as it indicates that the program does not satisfy the specifications and diagnosis of incorrectness or incompleteness should be performed.

3.2 Diagnosis by Proof

The existing proof methods for correctness and completeness are usually based on some kind of induction. Table 2 presents well-known sufficient conditions which can be used for program verification and diagnosis.

Property	**Definition**	**Implies**
$\mathcal{I}$ inductive for P	$S_P(\mathcal{I}) \subseteq \mathcal{I}$	P partially correct w.r.t. $\mathcal{I}$
$\mathcal{I}$ co-inductive for P	$\mathcal{I} \subseteq S_P(\mathcal{I})$	P complete* w.r.t. $\mathcal{I}$
$\mathcal{I}$ not inductive for P	$S_P(\mathcal{I}) - \mathcal{I} \neq \emptyset$	
$\mathcal{I}$ not co-inductive for P	$\mathcal{I} - S_P(\mathcal{I}) \neq \emptyset$	

Table 2: Set theoretic formulation of diagnosis by proof problems

In the table (*) stands for an additional requirement. A sufficient condition for completeness of P w.r.t. $\mathcal{I}$, requires not only co-inductiveness of $\mathcal{I}$ for P but also that S_P has a unique fixpoint. This last condition holds for a large class of programs (e.g., the acceptable programs in [AP93]).

Failures in proving the conditions may possibly indicate that the program has an error. An *incorrectness error* is a part of the program that is the reason for $S_P(\mathcal{I}) - \mathcal{I} \neq \emptyset$. An *incompleteness error* is a part of the program that is the reason for $\mathcal{I} - S_P(\mathcal{I}) \neq \emptyset$. The operator S_P in any kind of semantics is defined in terms of the constructs of the program P. Thus, it makes it possible to define precisely what is meant by the informal statement "is the reason". For CLP programs, an incorrectness error is a program clause and an incompleteness error is a program procedure (a set of the clauses defining a certain predicate symbol).

If the program is incorrect or incomplete, then it includes a corresponding error. One can try to make a proof that $\mathcal{I}$ is inductive (or co-inductive) w.r.t. the program. For an incorrect or incomplete program some constructs will be identified where the corresponding conditions cannot be proved. These constructs are possible error locations. As the conditions presented in Table 2 are not necessary, a fragment of the program localized as erroneous may or may not correspond to a bug in the program.

Example 3.1 *We show two examples for which a proof of partial correctness is impossible. In both cases the specification is not inductive for the program. In the first case the program is incorrect w.r.t. the specification. In the second, the program is correct but a correctness error is detected because of a too weak specification. The operator S_P is the immediate consequence operator T_P for logic programs.*

Consider the program P from Example 2.1 and the specification $\mathcal{I}_2$ from Example 2.4 (so the arguments of sorted *are required to be integer lists). An attempted correctness proof fails, $\mathcal{I}_2$ is not inductive w.r.t. P. The reason is the clause $sorted(X) \leftarrow X = [Y]$, as $sorted([a]) \in S_P(I_2)$ and $sorted([a]) \notin \mathcal{I}_2$. This clause is also the reason that the program is not partially correct w.r.t. $\mathcal{I}_2$.*

Consider the following CLP program Q, over the domain of integers. It is basically the program from Example 2.1 in which the new predicate order/2 *has been added and the second clause has been corrected.*

$$
\begin{array}{l}
sorted(X) \leftarrow X = [\,]. \\
sorted(X) \leftarrow X = [Y], Y > Z. \\
sorted(X) \leftarrow X = [H1|T1], T1 = [H2|T2], order(H1, H2), sorted(T1). \\
order(X, Y) \leftarrow X > Y.
\end{array}
$$

Assume that a partial specification requires the argument of sorted/1 *to be a list of integers. Nothing is required about predicate* order/2. *This means that, in our set-theoretical setting, $\mathcal{I}$ contains all the D-atoms of the form $order(X, Y)$ $(X, Y \in D)$ and all the atoms of the form $sorted(L)$, where L is a list of integers. Notice that Q is correct w.r.t. $\mathcal{I}$. However, $\mathcal{I}$ is not inductive w.r.t. Q (as $S_Q(\mathcal{I})$ contains for instance $sorted([a, 1])$). The third clause is the reason. Strengthening the specification for* order/2 *is necessary to obtain a correctness proof. We add a requirement that both arguments of* order/2 *are integers and obtain $\mathcal{I}'$, which is inductive w.r.t. Q.*

Note that the situation of weak correctness requirements presented above is equivalent to having an incomplete but correct program which presents a correctness error using conditions of Table 2 (or vice versa). However, the experience with type checking of logic programs (see, e.g., [AM94, HL94]) shows that failure in proving local validation conditions for a clause is often a good indication that the clause is erroneous.

3.3 Declarative Diagnosis

In contrast to diagnosis by proof, the declarative diagnosis concerns the case when a particular (test) computation does not satisfy a requirement.

We learn that a program P is incorrect (i.e., not partially correct w.r.t. $\mathcal{I}$) when we find out that it produces a result x such that $x \not\subseteq \mathcal{I}$. Such a result x is called an *incorrectness symptom*. Similarly, a program P is incomplete when it does not produce some expected result, in other words when there exists some $x \subseteq \mathcal{I}$ such that $x \not\subseteq [\![P]\!]$. Such x is called an *incompleteness symptom*.

Example 3.2 *In the program of Example 2.1 with the specifications of Example 2.4, note that $sorted([a]) \in [\![P]\!]$ but $sorted([a]) \notin \mathcal{I}_2$. Therefore, such an atom is an incorrectness symptom w.r.t. $\mathcal{I}_2$. If in that program the second clause was missing then $sorted([1]) \in \mathcal{I}_1$ would be an incompleteness symptom w.r.t. $\mathcal{I}_1$, since, without that clause, $sorted([1]) \notin [\![P]\!]$.*

Briefly, declarative diagnosis starts with a symptom of incorrectness (resp. insufficiency) and aims at localizing an erroneous fragment of the program. A declarative diagnoser localizes an error by comparing elements of the actual semantics involved in computation of the symptom at hand with user's expectations. The diagnoser will re-explore computations of symptoms obtained w.r.t. $\mathcal{I}$, and identify errors related to such symptoms, i.e., parts of the program which explain why $S_P(\mathcal{I}) \not\subseteq \mathcal{I}$ (resp. $\mathcal{I} \not\subseteq S_P(\mathcal{I})$). The erroneous fragment of the program localized in that way depends on the nature of S_P.

Example 3.3 *Consider (constraint) logic programming and its logical semantics. So $\mathcal{I}$ and $[\![P]\!]$ are interpretations over some domain. In the case of incorrectness, if there exists an x s.t. $x \in S_P(\mathcal{I})$ and $x \notin \mathcal{I}$ then there exists a clause $H \leftarrow B$ of the program P which is not valid in $\mathcal{I}$ (for some valuation, H is false and B is true). It can be proved that an incorrectness diagnoser finds such a erroneous clause for any incorrectness symptom. In the program of Example 2.1, with $\mathcal{I}_2$ as in Example 2.4, we have:*

$$
\begin{array}{rl}
T_P(\mathcal{I}_2) = & \{sorted([\,])\} \cup \{sorted([X]) \mid X \in D\} \cup \\
& \{sorted([X, Y|L]) \mid X, Y \textit{ are integers},\ X > Y,\ [Y|L] \textit{ is an integer list }\}
\end{array}
$$

in which $sorted([a])$ is included. The clause responsible for this symptom is the second one in the program.

In the case of incompleteness, if there exists an y s.t. $y \in \mathcal{I}$ and $y \notin S_P(\mathcal{I})$ then for each clause $H \leftarrow B$ of P if y is a value of H under some valuation ν (an instance of H) then $\nu(B)$ is false in $\mathcal{I}$. So the erroneous fragment found in this case is a set of clauses (which begin with the same predicate symbol).

In the process of diagnosing, the actual semantics of the program $[\![P]\!]$ is compared with the user's expectations $\mathcal{I}$. This is achieved by asking queries about elements of both $[\![P]\!]$ and $\mathcal{I}$ to an oracle. In practice the oracle is usually the programmer, although an executable specification may also be used (we will come back to this issue later).

Three families of queries are considered: one used in incorrectness error search and two used in incompleteness error search. A *universal query* asks whether a given subset Q of $[\![P]\!]$ is correct w.r.t. $\mathcal{I}$ (i.e. whether $Q \subseteq \mathcal{I}$). In the case of CLP, where $\mathcal{I}$ is a set of D-atoms, Q is usually the set of ground instances of a given constrained atom. An example universal query is:

Is $sorted([X,1]) \leftarrow X > 2$ correct?

The answer is YES, assuming that $\mathcal{I} = \{sorted(L) \mid L \text{ is a sorted integer list}\}$. Under the same assumption, the answer to the universal query about $sorted([X,1]) \leftarrow X \geq 0$ is NO.

An *existential query* asks whether a given set Q has an element in $\mathcal{I}$ (i.e. whether $Q \cap \mathcal{I} \neq \emptyset$). If Q is the set of ground instances of a constrained atom $A \leftarrow C$, then $Q \cap \mathcal{I} \neq \emptyset$ is equivalent to satisfiability of the formula $C \wedge A$ in the interpretation $\mathcal{I}$. Here is an example existential query (in which the constraint C is empty):

Is $sorted([X,Y])$ satisfiable?

A *covering query* asks if a given set Q' contains all the elements of a given set Q that are in $\mathcal{I}$ (so it asks whether $Q \cap \mathcal{I} \subseteq Q'$). It is a generalization of an existential query (when $Q' = \emptyset$). An example:

Do $\{sorted([2,1]), sorted([3,1])\}$ cover all correct instances of $sorted([X,1]) \leftarrow X{<}4$?

Query	**Answer**	**Definition**
Universal	yes	$Q \subseteq \mathcal{I}$
	no	$Q \not\subseteq \mathcal{I}$
Existential	yes	$Q \cap \mathcal{I} \neq \emptyset$
	no	$Q \cap \mathcal{I} = \emptyset$
Covering	yes	$(Q \cap \mathcal{I}) \subseteq Q'$
	no	$(Q \cap \mathcal{I}) \not\subseteq Q'$

Table 3: Set theoretic formulation of problems in a declarative diagnoser

Table 3 shows for all possible pairs of query/answer used in a declarative diagnoser the corresponding problem in a set theoretic setting.

4 Approximating the Intended Semantics

Using the exact intended semantics for automatic validation and diagnosis is in general not realistic, since the exact semantics can be only partially known and it is usually too inconvenient to express it formally. In this section we consider the debugging process in terms of *approximations* of the intended semantics. Approximations of the actual program semantics will be considered in the following sections.

An over-approximation of a value A (a "superset" if the semantic domain consists of sets), denoted A^+, satisfies $A \subseteq A^+$. Similarly, two other types of approximation are frequently considered, under- (or "subset") approximation, denoted A^-, $A^- \subseteq A$, and "existential" approximation, denoted $A^!$, $A^! \cap A \neq \emptyset$. In what follows, a prime symbol will used to distinguish an approximation A' from the exact value A.

Notice that if A_1^+ and A_2^+ are over-approximations of A then also $A_1^+ \cap A_2^+$ is an over-approximation of A. Moreover, it is a better approximation than either A_1^+ or A_2^+. A similar property holds for under-approximations w.r.t. $\cup$. However, existential approximations do not enjoy this property.

Example 4.1 *Consider the CLP program given in Example 2.1, and its specifications in Example 2.4. We have that $\mathcal{I}_1$ is an under-approximation of the intended semantics $\mathcal{I}$, and $\mathcal{I}_2$ is an over-approximation of it. Therefore, $\mathcal{I}_1$ (resp. $\mathcal{I}_2$) is a specification of kind $\mathcal{I}^-$ (resp. $\mathcal{I}^+$), and can be used in proving properties w.r.t. $\mathcal{I}$. A different thing is that while trying to prove properties w.r.t. $\mathcal{I}_1$ (resp. $\mathcal{I}_2$) we may try to use also approximations of the form $\mathcal{I}_1^-$ (resp. $\mathcal{I}_2^+$) or $\mathcal{I}_1^+$ (resp. $\mathcal{I}_2^-$).*

We now discuss the use of approximations in program diagnosis.

4.1 Replacing the Oracle in Declarative Diagnosis

As seen in Section 3.3, in declarative diagnosis the existence of an oracle is assumed and the user is repeatedly asked questions about the intended semantics of the program. An idea is then to provide the system with (an approximation of) the intended semantics which can be used to automatically answer some of the oracle queries. When no sufficient conditions for a given query are satisfied, then the query cannot be answered automatically and the answer has to be provided by the user.

It is very seldom the case that there exists a formal specification $\mathcal{I}$ which completely describes the user's intention. Even less realistic is to expect that there exists such an executable specification. However, it is feasible to have formal/executable specifications which are approximations $\mathcal{I}^+$, $\mathcal{I}^-$ or $\mathcal{I}^!$ of the intended semantics. Such approximate specifications for declarative debugging of logic programs were introduced in [DNTM89], where four kinds of approximations were used. In our terminology those approximations were $\mathcal{I}^-$, $(\overline{\mathcal{I}})^!$, $\mathcal{I}^!$ and $\overline{\mathcal{I}^+}$ or, equivalently, $(\overline{\mathcal{I}})^-$ (where $\overline{S}$ denotes the complement of set S). That paper reported on experiments performed with a prototype implementation which was used to partially automate the answering of queries (except covering queries). User's answers (to the queries not answered automatically) are stored as an executable partial specification, which can then be used if the query is repeated. Actually, in some cases it can also be used to answer other queries. Table 4 presents a series of sufficient conditions which can be used by a declarative diagnoser to automatically answer some of the questions and avoid asking the user.

Name	Property	Sufficient condition
Universal	$Q \subseteq \mathcal{I}$	$Q \subseteq \mathcal{I}^-$
	$Q \not\subseteq \mathcal{I}$	$Q \not\subseteq \mathcal{I}^+$, or $Q \cap \mathcal{I}^+ = \emptyset \wedge Q \neq \emptyset$
Existential	$Q \cap \mathcal{I} = \emptyset$	$Q \cap \mathcal{I}^+ = \emptyset$
	$Q \cap \mathcal{I} \neq \emptyset$	$Q \cap \mathcal{I}^- \neq \emptyset$, or $Q \subseteq \mathcal{I}^-$, or $\mathcal{I}^! \subseteq Q$

Table 4: Sufficient conditions for oracle queries

Example 4.2 *Assume the query $\{sorted([a])\} \subseteq \mathcal{I}$ posed during incorrectness diagnosis. An approximation $\mathcal{I}^+$ containing all the atoms of the form $sorted(V)$ where V is an integer list (such as $\mathcal{I}_2$ of Example 2.4) is sufficient to obtain a negative answer.*

5 Approximating the Actual Semantics

The methods of program analysis allow computing approximations of the actual semantics $[\![P]\!]$, thus automating validation of programs w.r.t. a priori chosen properties.

One of the most successful techniques for approximating the actual semantics of a program is *abstract interpretation* [CC77]. In this technique a program is interpreted over a non-standard domain called *abstract* domain D_α and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program is computed (or approximated) by replacing the operators in the program by their abstract counterparts.

The idea of using abstract interpretation for validation and diagnosis is not new. Its use for debugging of imperative programs has been studied by Bourdoncle [Bou93], and for debugging of logic programs by Comini et al. [CLMV96b]. Both approaches focus on some specific semantics and specific programming languages. It has also been used in abstract assertion checking proposed in [BCHP96]. This section outlines the use of abstract interpretation for verification and diagnosis in a general setting of arbitrary fixpoint (set) semantics. For the time being, we assume that specifications are written as $\mathcal{I}_\alpha$ (i.e., the abstract domain is used as the language to write specifications). Thus, we discuss proving properties w.r.t. I_α, and only approximations of the actual model are considered.

5.1 Abstract Interpretation

An abstract semantic object is a finite representation of a, possibly infinite, set of actual semantic objects in the concrete domain (D). The set of all possible abstract semantic values represents an *abstract domain* (D_α) which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle D, \subseteq\rangle$ and abstract $\langle D_\alpha, \subseteq_\alpha\rangle$ domains. The concrete and abstract domains are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, such that

$$\forall x \in D: \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha: \ \alpha(\gamma(y)) = y. \tag{1}$$

Note that in general $\subseteq_\alpha$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha: \ \lambda \subseteq_\alpha \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$), and is not equal to set inclusion. In an abuse of notation, however, we will usually write $\subseteq$ both for the concrete and abstract domain. Similarly, the operations of *least upper bound* ($\cup_\alpha$) and *greatest lower bound* ($\cap_\alpha$) mimic those of D in some precise sense. Again, in an abuse of notation, we will use $\cup$ and $\cap$, respectively (although they are in general not equal).

By monotonicity, the mappings α and γ (denoted f in what follows) satisfy:

$$x \subseteq y \ \Rightarrow \ f(x) \subseteq f(y). \tag{2}$$

We will also assume in some cases the following properties for α and γ:

$$f(x) \cap f(y) = \emptyset \Rightarrow \ x \cap y = \emptyset \quad \text{and} \quad f(x \cap y) = f(x) \cap f(y). \tag{3}$$

The abstract domain D_α is usually constructed with the objective of computing approximations of the semantics of a given program. Thus, all operations in the abstract domain also have to abstract their concrete counterparts. In particular, if the semantic operator S_P can be decomposed in lower level operations, and their abstract counterparts are locally correct w.r.t. them, then an abstract semantic operator S_P^α can be defined which is correct w.r.t. S_P. This means that $\gamma(S_P^\alpha(\alpha(x))$ is an approximation of $S_P(x)$ in D, and consequently, $\gamma(lfp(S_P^\alpha))$ is an approximation of $[\![P]\!]$. We will denote $lfp(S_P^\alpha)$ as $[\![P]\!]_\alpha$. The following relations hold:

$$\forall x \in D: \ \gamma(S_P^\alpha(\alpha(x)) \supseteq S_P(x) \tag{4}$$

$$\gamma([\![P]\!]_\alpha) \supseteq [\![P]\!] \ \text{ equivalently } \ [\![P]\!]_\alpha \supseteq \alpha([\![P]\!]). \tag{5}$$

An abstract operator S_P^α is said to be *precise*, if instead it satisfies that

$$\gamma([\![P]\!]_\alpha) = [\![P]\!] \text{ equivalently } [\![P]\!]_\alpha = \alpha([\![P]\!]). \tag{6}$$

Note that the construction presented allows obtaining over-approximations of $[\![P]\!]$. When (1) holds, the construction is termed a Galois insertion. If $\subseteq$ is used in (1) instead of $\supseteq$, we obtain a dual construction, termed a reversed Galois insertion. The dual relations of (4) and (5) also hold in this case.

In practice, the abstract domains should be sufficiently simple to allow effective computation of semantic approximations of programs. For example, Herbrand interpretations of some alphabet may be mapped into an abstract domain where each element represents a typing of predicates in some type system. For a given program P the abstract operator S_P^α would allow then to compute a typing of the predicates in the least Herbrand model of P.

Example 5.1 *A simple example of abstract interpretation in logic programming can be constructed as follows. The concrete semantics (least Herbrand model) of a program P is $[\![P]\!] = lfp(T_P)$. So the concrete domain is $D = \wp(B_P)$ (where B_P is the Herbrand base of the program).*

We consider over-approximating $[\![P]\!]$ by the set of "succeeding predicates", i.e. those whose predicate symbols appear in $[\![P]\!]$. A possible abstraction is as follows. The abstract domain is $D_\alpha = \wp(B_P^\alpha)$, where B_P^α is the set of predicate symbols of P. Let $pred(A)$ denote the predicate symbol for an atom A. We define the abstraction function:

$$\alpha : D \to D_\alpha \;\; \textit{such that} \;\; \alpha(I) = \{pred(A) \mid A \in I\}.$$

The concretization function is defined as:

$$\gamma : D_\alpha \to D \;\; \textit{such that} \;\; \gamma(I_\alpha) = \{A \in B_P \mid pred(A) \in I_\alpha\}.$$

For example,

$$\alpha(\{p(a,b), p(c,d), q(a), r(a)\}) = \{p,q,r\}$$
$$\gamma(\{p,q\}) = \{p(a,a), p(a,b), p(a,c), \ldots, q(a), q(b), \ldots\}.$$

Note that $(D_\alpha, \gamma, D, \alpha)$ is a Galois insertion. The abstract semantic operator $T_P^\alpha : D_\alpha \to D_\alpha$ is defined as:

$$T_P^\alpha(I_\alpha) = \{pred(A) \mid \exists\, (A \leftarrow B_1, \ldots, B_n) \in P \; \forall i \in [1,n] : pred(B_i) \in I_\alpha\}.$$

Since D_α is finite and T_P^α is monotonic, the analysis (applying T_P^α repeatedly until fixpoint, starting from $\emptyset$) will terminate in a finite number of steps n and $[\![P]\!]_\alpha = T_P^\alpha \!\uparrow\! n$ approximates $[\![P]\!]$.

For example, for the following program P,

$$p(X) \leftarrow q(X). \quad r(X). \quad p(X) \leftarrow r(X). \quad s(X). \quad t(X) \leftarrow l(X). \quad m(X). \quad q(a). \quad q(b). \quad r(a). \quad r(c).$$

we have $B_P^\alpha = \{p,q,r,s,t,l,m\}$, and:

$$T_P^\alpha(\emptyset) = \{q,r\} \quad T_P^\alpha(\{q,r\}) = \{q,r,p\} \quad T_P^\alpha(\{q,r,p\}) = \{q,r,p\}$$

So $T_P^\alpha \!\uparrow\! 2 = T_P^\alpha \!\uparrow\! 3 = \{q,r,p\} = [\![P]\!]_\alpha$

5.2 Abstract Diagnosis

The technique of abstract diagnosis [CLMV96b, CLMV96a] is based on the use of *observables* which correspond roughly to the abstraction functions α used in abstract interpretation with some additional properties. Observables (in a similar way to semantics) allow extracting the properties of interest from the execution of a goal, while hiding details which are not relevant. The intended semantics with respect to the observable α is denoted $\mathcal{I}_\alpha$ and is assumed to be an exact description.

Abstract diagnosis searches for incorrectness and incompleteness errors as defined in Section 3.2, using the sufficient conditions given in Table 2. The semantic operator S_P is replaced by S_P^α, in a similar way to abstract interpretation. Unlike abstract interpretation, no fixpoint computation is needed and $lfp(S_P^\alpha)$ is not computed.

Two different kind of observables are considered in [CLMV96a]. *Complete* observables provide stronger results but are often not practical because the specification of the intended semantics $\mathcal{I}_\alpha$ is infinite and diagnosis would not terminate. Such complete observables correspond to the precise abstract operators of Section 5.1. The second kind of observables considered in [CLMV96a] are called *approximate* observables and their corresponding operator S_P^α is correct but not precise (as is usually the case in abstract interpretation).

5.3 Validation using Abstract Interpretation

Abstract diagnosis localizes suspected program constructs following the diagnosis by proof principle. The proof attempt may succeed in which case the program satisfies the requirement $\mathcal{I}$ (expressed as $\mathcal{I}_\alpha$), and abstract diagnosis works as validation.

An alternative way of validation is to compute abstract approximations $[\![P]\!]_\alpha$ of the actual semantics of the program $[\![P]\!]$ and then use the definitions given in Table 1 instead of the sufficient conditions of Table 2 (on which abstract diagnosis is based). This is reasonable if one considers that usually program analyses are performed in any case to use the information inferred for optimizing the code of the program.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program should be done in the same domain. Thus, for comparison we need in principle $\alpha([\![P]\!])$. However, using abstract interpretation, we instead compute $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$, and can be compared with $\mathcal{I}_\alpha$. We will use the notation $[\![P]\!]_{\alpha^+}$ to represent that $[\![P]\!]_\alpha \supseteq \alpha([\![P]\!])$. $[\![P]\!]_{\alpha^-}$ indicates that $[\![P]\!]_\alpha \subseteq \alpha([\![P]\!])$. Table 5 gives sufficient conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$ which can be used when $[\![P]\!]$ is approximated.

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

Table 5: Validation problems using approximations

The following conclusions can be drawn from Table 5. Analyses which use a Galois insertion (α^+, γ^+), and thus over-approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$) are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred by the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. Note that it will only be possible to prove completeness if the abstraction is precise. According to Table 5 only $[\![P]\!]_{\alpha^-}$ can be used at this end, and in the case we are discussing $[\![P]\!]_{\alpha^+}$ holds. Thus, the only possibility is that the abstraction is precise.

On the other hand, if a reversed Galois insertion is used (α^-, γ^-), and then analysis under-approximates the actual semantics (the case denoted $[\![P]\!]_{\alpha^-}$), it will be possible to prove completeness and incorrectness. Partial correctness and incompleteness can only be proved if the analysis is precise.

Note that the results obtained for direct Galois insertions (α^+, γ^+) are in essence equivalent to the ones presented for abstract diagnosis [CLMV96a] for *approximate* observables. In the case of precise abstractions, also completeness may be derivable, and this corresponds to the complete observables of [CLMV96a].

Example 5.2 *If the abstract interpretation tells that in $[\![P]\!]_{\alpha+}$ the type of a predicate p with just one argument position is intlist and the user has declared it in $\mathcal{I}_\alpha$ as list, then under some natural assumptions about ordering in the abstract domain we conclude that $[\![P]\!]_{\alpha+} \subseteq \mathcal{I}_\alpha$, i.e., the program is correct w.r.t. the declared $\mathcal{I}_\alpha$ (or more precisely w.r.t. $\gamma(\mathcal{I}_\alpha)$). However, the program may still be incorrect w.r.t. the precise intention $\mathcal{I}$, which is not given by the declaration.*

Example 5.3 *Assume now that $[\![P]\!]_{\alpha+} \not\subseteq \mathcal{I}_\alpha$. We cannot conclude that P is correct w.r.t. $\mathcal{I}_\alpha$. We cannot conclude the contrary either. For example if the abstract interpretation tells that the type of the predicate p with one argument position is list while the user declares it as intlist then P may still be correct w.r.t. the declaration. This can be due to the loss of accuracy introduced by the abstraction. In any case it may be desirable to localize a fragment of the program responsible for this discrepancy. A more careful inspection would then be needed to check whether the fragment is erroneous w.r.t. the declaration, or not.*

If analysis information allow us to conclude that the program is incorrect or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, a diagnosis should be performed to locate the program construct responsible for the symptom. We are studying the possibility of using for that purpose the conditions in Table 2, in a similar way as done in abstract diagnosis [CLMV96b].

6 Towards an Integrated Validation and Diagnosis Environment

In the previous sections we have addressed the problem of validation and diagnosis of a program with respect to incomplete requirements. We have hopefully contributed to clarifying how known verification and debugging techniques can be combined to support the process of program development, specially in the case in which approximations are used. This final section discusses the design of an environment integrating validation and diagnosis tools making an extensive use of semantic approximations. The development of an environment of this kind is one of the objectives of the DiSCiPl project. The design of the language for expressing semantic approximations is an essential part of this task since the language is crucial for the user interface for the integration of different tools. We discuss some possibilities regarding the design of this language, some aspects of the debugging process, and the structure of the environment itself.

6.1 Assertion Languages

An *assertion* is a linguistic expression which uniquely identifies an element x of the semantic domain D. The role of assertions is to express approximations of the program semantics, thus we expect that an assertion includes also an indication concerning the kind of approximation it is intended to express. The approximations we propose to allow in assertions are again those introduced in Section 4, i.e., $\{+, -, !\}$. In practice, assertions can be used to describe not only the intended semantics but also the actual semantics of the program (an example of the latter is the use of assertions to express the result of program analysis in [BCHP96]).

We now consider several possible choices for the language of assertions. Notice that different kind of semantics may be used for validation and for debugging, thus the choices discussed below are parameterized by the semantic domain used.

Note that semantic values in our setting are sets. Thus the assertions have to describe sets. One can adopt for that purpose some of the standard notations. For example formulae of first-order predicate calculus with appropriate interpretation may be used for that purpose.

When using a particular abstract domain for program analysis for abstract debugging, or for validation, it may be more convenient to introduce a specialized language referring to the elements of this domain. The language of types inferred by some abstract interpretation is a typical example of this kind. An assertion of such a language corresponds to an element of the abstract domain. The concretization function maps it to the concrete domain. An advantage of using assertions specialized for a given abstract domain is that they facilitate interaction between the user and the tools. For example, when working with types it can be more natural to express and compare types inferred by the system and types declared by the user, rather than types obtained from assertions about the program which were not expressed in terms of types. It is also possible to consider assertions written in a different abstract domain than the one used for comparison. The limitation when using the abstract domain(s) is that the semantic objects which can be described are limited to those which are present as elements of the different abstract domains available in the tools.

Another alternative is to express assertions in the (subset of) the same programming language i.e., an assertion for a program P is another program A, thus the semantic object indicated by A is $[\![A]\!]$. For example, Prolog programs are used as assertions in the tool discussed in Section 4.1, and the restricted set of regular programs [GdW94] can be used as type assertions in the CIAO system. Assertions in the form of programs can be seen as executable specifications. Such a program may be used to compute elements of the set specified by it. This technique is especially useful in higher level languages such as CLP, where the result of a computation may be a finite representation of an infinite set. An additional advantage of using programs as assertions is that the assertions themselves can be used as run-time tests in the cases in which the desired properties cannot be proven automatically (see Section 6.3).

Notice that assertions written in different assertion languages refer to the same semantic domain. Therefore it may be desirable to develop techniques of combining them for purposes of specification.

Example 6.1 *In the case of the D-model semantics a language of D-constraints including some basic constraints and closed under conjunction and existential quantification might be an appropriate candidate. Checking the verification conditions would then require a constraint solver capable of checking satisfiability and entailment. For the program of Example 2.1, a possible specification is the following:*

$$A_1 = (\ \{\ sorted(X)\ :-\ list(X).\ \ list([]).\ \ list([\_|Y])\ :-\ list(Y).\ \},\ +).$$

$$A_2 = (\ \{\ sorted([X,Y])\ :-\ X > Y.\ \},\ -).$$

which are obviously valid assertions describing over- and under-approximations of the user's intention, respectively.

6.2 Some Practical Aspects of the Debugging Process

An important aspect of debugging is that in practice the process of program construction is often iterative, and the iterations update incrementally not only the program but also the requirements. This is related to the observation that user's expectations concerning a program are rarely fully described. At each stage of development we have a (possibly empty) subset approximation $\mathcal{I}^-$ of the intended semantics and a (possibly empty) superset approximation $\mathcal{I}^+$[3] which together represent the specification. The program in hand should be complete w.r.t. $\mathcal{I}^-$ and partially correct w.r.t. $\mathcal{I}^+$. In the previous sections we mentioned some well-known proof methods used for checking that.

If the proof fails, the failure points to some fragments of the program, which may possibly be erroneous. The failure may be due to: (1) an error in the program causing violation of the specification at hand, (2) the specification is too weak, or (3) incompleteness of the prover. Note that if an error exists then it can only be due to the fragments identified. The user should inspect them in order to identify the reason of failure. If the user identifies that the reason is an error

[3]The other kinds of approximations may also be present but, for simplicity, we will consider these two in this discussion.

then the program has to be corrected. If, on the other hand, the user does not identify an error then alternatively it may be possible to strengthen the specification in such a way that a proof can be achieved.

If the proof succeeds we may: (1) stop the development process, or (2) update the specification. In particular, the latter is needed if the behavior of the program w.r.t. the first specification is not acceptable and the user wants to clarify why.

Note that even if the proof succeeds some bugs may still be hidden in the program, as specifications are partial. For example, if the techniques presented in Section 5 are used, some bugs may not be captured by the abstract semantics. Thus, if during testing or execution of the program some unexpected behaviour is found, diagnosis should start for it. The well-known technique of declarative diagnosis is then applicable, which, as we have seen, can also rely on approximations of the intended semantics.

6.3 Which tools are needed

We believe that an integrated environment incorporating the techniques described so far (as well as other techniques, such as procedural debugging and visualization, which are beyond the scope of this paper) can be of great help in speeding up the code development process. In this section we propose some tools to be included in the environment. The intention is to detect bugs as early as possible, i.e., during compilation or even editing. This can only be achieved by (semi-) automatic analysis of the (not necessarily completely developed) program in the presence of some (approximate) specifications. An example of such techniques is type checking, which proved to be useful for that purpose. Our approach puts a framework for working with properties that may be more general than classical type systems.

The common integrating concept for the tools proposed is the notion of semantic approximation which is involved in

- describing user's intentions,
- program analysis,
- comparing the results of program analysis with the user's intentions,
- verification,
- debugging.

The fundamental technique mentioned in this context is that of abstract interpretation which allows for automatic synthesis of semantic approximations, for abstract verification and for abstract debugging.

To support the above mentioned activities we may need the following tools:

- A program analyzer: it takes the program and the selected abstract domain(s) and generates an approximation of the actual semantics of the program. In the case of CLP programs standard analysis techniques can be used for this purpose.
- An assertion translator: if the language for assertions is the underlying programming language or an abstract domain different from that used internally by the tool, this translator is in charge of transforming the intended semantics into the abstract domain to be used by the analyzer. An intelligent translation scheme would be able to select the best among a set of abstract domains depending on the requirements expressed by the user in the intended model.
- A comparator: it would compare the user requirements and the information generated by the analysis. It can produce three different results:
  - The requirement is verified.

– The requirement does not hold. An *abstract symptom* has been found and diagnosis should start.

– None of the above. We cannot prove that the requirement holds nor that it does not hold. Run-time tests could be introduced which would make sure that the requirements hold. Clearly, this introduces an important overhead and could be turned on only during program testing.

- A diagnoser based on abstraction: the diagnoser tries to localize the program construct responsible for the abstract symptom. It would use algorithms based on the sufficient conditions of Table 2. Thus it will locate possible error sources.
- A declarative (concrete) diagnoser: it would be used once all abstract symptoms have been diagnosed and eliminated from the program in order to underpin all subsequent bugs in the program which appear during program testing and execution. As in Section 4.1, the program would store approximations of the intended semantics to avoid asking the user whenever the question can be solved using such approximations.

Partial prototypes of the component tools are mentioned above are currently being developed. For example, an assertion language, translator, analyzer, and comparator has been incorporated in the CIAO system which works on the domains of moded types, definiteness, freeness, and grounding dependencies for CLP programs.

References

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CLMV96a] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. Submitted for publication, 1996.

[CLMV96b] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In (H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[Fer87] G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[GHB+96] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 13/20:503–581, 1994.

[Sha82] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.