A Symbolic Constraint Solving Framework for Analysis of Logic Programs

C.R. Ramakrishnan
I.V. Ramakrishnan
Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400.

R.C. Sekar
Bellcore
445, South Street
Morristown, NJ 07960.

Abstract

Abstract Interpretation of logic programs using symbolic constraints has attracted a lot of attention lately since such approaches can yield very elegant formulation of many analyses. Moreover, the performance of an analysis that uses constraints to represent program properties and symbolic constraint solving techniques to infer them, can be largely insensitive to domain size. However, implementations of these techniques must balance the conflicting requirements of (a) providing efficient constraint solving algorithms for specific constraints, and (b) being general enough to deal with a large class of constraints. We overcome the apparent conflict by implementing the analysis and constraint solving operations in multiple layers such that the top most layer is completely generic. The lower layers become increasingly specialized for a particular analysis. Each layer provides a set of operations for the higher layer using the operations provided by the lower layers. Such layering enables us to choose the most efficient algorithms appropriate for that layer, independent of the other layers. The critical aspect of this framework is the identification of interfaces between the layers that enables us to modularize not only our algorithms and implementations, but also the proof efforts. A prototype implementation of our framework shows that it scales very well to large domains, and furthermore, compares favorably with the existing implementations of other analysis methods.

Contact Author : C.R. Ramakrishnan
e-mail : `cram@cs.sunysb.edu`
Phone : (516) 632 8470
Fax : (516) 632 8334

This main paper is 13 pages long including figures and tables. The appendix contains supplementary reference material and is provided for the convenience of the referees.

1 Introduction

Abstract interpretation, introduced by Cousot and Cousot in [8], provides a foundation for analysis of programs. Logic program analysis using abstract interpretation has received considerable attention (e.g., see [9]). Abstract interpretation based analysis interprets the program using a nonstandard semantics, where the concrete domain of values is replaced by an abstract domain of descriptions of values, and each operator is replaced by the corresponding nonstandard interpretation. Top-down analyses (e.g., [1]) compute the set of abstract substitutions for each variable at each call site; bottom-up analyses compute the abstract success set of each predicate. Thus, in abstract interpretation, we compute a set of tuples of substitutions that describes a relation (*i.e.*, the property of interest) over the abstract domain. Implementations of abstract interpretation typically compute the program property by enumerating the tuples in this relation (e.g., see [3, 5]).

Note that the efficiency of enumeration-driven implementations is dependent on the size of the abstract domain. For example, consider a predicate $p(K, K)$ interpreted over a domain $D = \{d_1, \ldots, d_n\}$. Enumerative methods represent the relation as the set $\{(d_1, d_1), \ldots, (d_n, d_n)\}$. The size of this representation can become large for large domains, particularly for domains obtained as a product of other domains [6]. Alternatively, the same relation can be represented symbolically using the constraint $(X_1 = X_2)$, with its size unaffected by changes to the domain. Thus methods based on symbolic representation can be less sensitive to domain size.

The idea of using symbolic constraints to represent relations was first proposed by Wegbreit [23] for analysis of flow chart programs. In the context of logic programs, the use of symbolic constraints was formally studied by Codognet and File [4] and Giacobazzi et al [13]. They compute the relation using a fix-point algorithm that is parameterized by operations on the constraints. However, the design of efficient algorithms for actually performing these constraint operations was not addressed. Corsini et al, in [5], present an efficient static analyzer where the relations are represented by symbolic constraints in the Toupie language. The constraint solving technique used in this work is based on Bryant's Decision Diagrams [2] which compactly represent the relations, leading to a very efficient implementation. However, this representation is still enumerative thereby making the implementation domain dependent. Analyses of Jacobs and Langen [14], Muthukumar and Hermenegildo [20] and others to detect sharing among program variables, can be viewed as using symbolic constraint solving techniques. In addition to providing a formal basis for analysis of sharing in an abstract interpretation framework, these works also present algorithms for operations on the sharing constraints. However, their works do not explore the effect of adding other constraints on this algorithm, or the effect of dealing with abstract domains other than those of groundness and freeness. We address the above issues in this paper and develop a modular framework for implementing bottom-up abstract interpretation of logic programs using symbolic constraint solving techniques.

1.1 Overview

The salient feature of our work is that it proposes a solution to the conflicting problems of

- providing constraint solving algorithms, which necessarily requires the method to focus on the specifics of the constraints, and
- providing a general method that is capable of dealing with a large class of constraints.

We overcome this apparent conflict by splitting the constraint solving process into several *layers*. Such layering permits us to modularize our algorithms so that dependence on specific characteristics

Layer	Requires	Provides
Generic Layer	Simplify ($\mathcal{S}$), project ($\mathcal{P}$), implication ($\mathcal{I}$), base constraints	Fix point iteration, composition
Boolean Simplification Layer	product ($*$), quotient ($/$), project conjuncts ($\mathcal{P}_\wedge$)	$\mathcal{S}$, $\mathcal{P}$, $\mathcal{I}$
Atomic Constraints Layer	*glb*, *subset*, *superset*	$*$, $/$ and $\mathcal{P}_\wedge$
Domain Layer	nothing	*glb*, *subset*, *superset*, base constraints

Figure 1: Structure of the Framework

of constraints are localized. The layers in our framework are shown in Figure 1. The top most layer is generic and does not even fix a representation for the constraints. Lower layers successively specify the form of these constraints and become increasingly specialized for a particular analysis. Each layer provides a set of operations for the higher layer. These operations are parameterized with respect to the operations provided by the lower layer. However, the algorithms in any layer are based only on those aspects of the constraint representation that are fixed by that layer. Because of this, the implementation of any layer is unaffected by changes to the layers above or below it.

The critical aspect of this framework is the identification of the interfaces between the layers. An interface is characterized by a set of operations, along with a set of correctness requirements for these operations. We establish the correctness of algorithms specified in a layer (which are in turn the requirements of the upper layer) based only on the correctness requirements on the lower layer. This enables us to modularize not only the algorithms and implementations, but also the proof efforts. In the following we briefly describe the overall structcture of the framework.

The top layer of the framework, called the *generic layer*, operates on programs and computes program properties bottom up, using composition and fix point operations. It relies on the lower layers to define the representation of these properties and provide three operations, namely, *simplify*, *project* and *implication*. *Simplify* is used to maintain these properties in a canonical form; *project* is used to eliminate intermediate variables from properties; and *implication* is used to check whether a fix point has been reached. In terms of functionality this layer corresponds to the fix-point algorithms in [4, 23] and the GAIA [17].

The *boolean simplification layer* specifies the form of the properties to be disjunctive normal formulae (DNF's) over *atomic constraints*. (The form of atomic constraints is, in turn, specified by lower layers.) It realizes the *simplify* and *implication* operations on DNF's through a set of operations (called *quotient* and *product*) on atomic constraints. The *quotient* operation, denoted by '/', is a generalization of implication so that C_1/C_2 returns the set of additional constraints under which C_2 implies C_1. The *product* operation, denoted by '$*$', returns the conjunction of two constraints in canonical form.

The third layer, called the *atomic constraints layer*, fixes the form of atomic constraints. In this paper, we have chosen two types of atomic constraints: the equality constraint of the form $X = t$, where t is a term (possibly containing other variables) whose depth is bounded; and a *membership* constraint of the form $X \in d$, which captures the notion that X takes substitution from a subset of the Herbrand base denoted by d. The latter constraint is designed to capture the notion of abstract substitution, as in abstract interpretation based analyses. This layer provides the operations $*$ and

/ for simplifying atomic constraints and the projection on conjuncts, based on the domain-specific operations called *glb, subset* and *superset.* These operations are provided by the *domain layer.* For simplicity of presentation, we restrict the formulation of the framework to domains that possess finite ascending chain property. The framework can be easily extended to general domains using widening operations [12], as discussed in section 8.

The rest of this paper is organized as follows. The four layers in our framework are described in sections 2 through 5. Following this, we present the results of our prototype implementation. These results compare favorably with the results of two well known implementations, namely, Toupie [5], that performs bottom-up abstract interpretation and the GAIA [16], that performs top-down abstract interpretation. The results also show that our method is largely insensitive to domain size. For instance, a three-fold increase in the domain size resulted in an average increase of 10% in the analysis time. In contrast, in the system based on Toupie solver increases in domain size leads to (more than) proportionate increase in analysis times. In section 7 we compare the accuracy of our method with other analysis methods, and indicate possible extensions to our framework. Details that may aid the refereeing process but are not central to the paper are given in the appendix.

Notation The symbols of our language are drawn from three mutually exclusive sets: *variables* denoted by X, Y, Z; *function symbols* denoted by a, b, c; and *predicate symbols*, denoted by p, q, r. The symbols may appear with or without subscripts. Lists and sets of variables are denoted by symbols $\overline{X}, \overline{Y}, \overline{Z}$. Constraints (denoted by φ and ψ) are built using conjunction and disjunction over *atomic constraints* of the form $C(X_1, \ldots, X_n)$, where C is the name of the atomic constraint and $X_1, \ldots, X_n$ are variables. Thus constraints are (negation-free) first order formulae with free variables. Logical implication (denoted by '$\Rightarrow$'), defined as usual in terms of substitutions, forms a partial order over the set of constraints.

2 Generic Layer

The generic layer is parameterized with respect to the base constraints (*i.e.*, constraints on built-in predicates) and the operations of simplification ($\mathcal{S}$), projection ($\mathcal{P}$) and implication ($\mathcal{I}$). The properties of predicates are computed bottom up based on these operations. We restrict our attention to properties that are valid when a predicate succeeds. Properties that are dependent on some execution order can be obtained using transformation techniques such as Magic templates [21].

For every predicate $p(\overline{X})$ in the program we associate a constraint $p^{\#}(\overline{X})$ such that every substitution that satisfies p also satisfies $p^{\#}$. Hence $p^{\#}$ is a *necessary* condition for p to succeed and therefore can be viewed as a property[1] of p. Given $r^{\#}$ for every built-in predicate r, the aim of the analysis is to compute $p^{\#}$ for every p defined in the program. Let a predicate p be defined by clauses of the following form:

$$\begin{array}{rcl} p_1(\overline{X}) & \leftarrow & q_1^1(\overline{X}_1^1), \ldots, q_{m_1}^1(\overline{X}_{r_1}^1). \\ \vdots & \vdots & \vdots \\ p_n(\overline{X}) & \leftarrow & q_1^n(\overline{X}_1^n), \ldots, q_{m_n}^n(\overline{X}_{r_n}^n). \end{array} \qquad (1)$$

[1]Properties that depend on sufficient conditions of the predicates can also be obtained in our framework by essentially reversing the direction of implication.

Then the constraint $p^{\#}$ is given by the following equations:

$$p_j{}^{\#} = \mathcal{P}^{\overline{X}}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j}(q_k^j)^{\#}))$$

$$p^{\#} = \mathcal{S}_{\vee}(\bigvee_{j=1}^{n} p_j{}^{\#})$$

The functions $\mathcal{S}_{\wedge}$ and $\mathcal{S}_{\vee}$, henceforth referred to collectively as $\mathcal{S}$, simplify a given constraint and maintain it in a canonical form. The operation $\mathcal{P}$ projects the given constraint onto $\overline{X}$, the head variables of p. For each built-in predicate r (e.g., '$<$' and 'is') we have $r^{\#}(\overline{X}) = \varphi_r(\overline{X})$ where φ_r is the base constraint fixed by the particular analysis.

The above rules are used directly to compute $p^{\#}$ if p is nonrecursive. Otherwise $p^{\#}$ is computed by a fix-point iteration procedure as the limit of the sequence $p^{\#,0}, p^{\#,1}, \ldots$ defined as follows[2]:

$$p^{\#,0} = False$$

$$p_j{}^{\#,i} = \mathcal{P}^{\overline{X}}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j}(q_k^j)^{\#,i-1}))$$

$$p^{\#,i} = \mathcal{S}_{\vee}(\bigvee_{j=1}^{n} p_j{}^{\#,i})$$

The fix point is identified using the implication operator $\mathcal{I}$ that tests if one constraint implies the other. In order to stop iterations as soon as a fix point is reached, the implication test must be complete. However, many constraint domains lack independence of negated constraints (see Jaffar and Maher [15]); *i.e.*, a finite disjunction φ may imply a conjunction ψ, but no individual conjunct in φ may imply ψ. In such domains, complete implication tests are notoriously hard since whole disjunctions have to be considered at once. Hence we relax the conditions on $\mathcal{I}$ and only demand that it be *sound*. But, to ensure termination, we require the existence of a partial order $\sqsubseteq$ such that $\mathcal{S}$ and $\mathcal{P}$ are monotonic in $\sqsubseteq$ and $\mathcal{I}$ is complete with respect to $\sqsubseteq$; *i.e.*, $\forall \varphi_1, \varphi_2 \quad \varphi_1 \sqsubseteq \varphi_2 \Rightarrow \mathcal{I}(\varphi_1, \varphi_2)$. Note that the particular order $\sqsubseteq$ is fixed in the lower layers; all we need here in this layer is a guarantee that this property holds.

Clearly, with an incomplete implication test we may not stop iterations as soon as a fix point is reached. Nevertheless, to preserve the genericity of this layer we leave it to the lower layers to make the tradeoff between (possibly) longer iterations and an expensive test for implication.

Proof Obligations In order to prove the soundness and termination of this layer we assume the following properties on the base constraints and $\mathcal{S}$, $\mathcal{P}$ and $\mathcal{I}$.

Requirement 2.1 (Soundness)
a. *The simplification, projection, and implication operators $\mathcal{S}$, $\mathcal{P}$ and $\mathcal{I}$ are* sound*; i.e., for every constraint φ, $\varphi \Rightarrow \mathcal{S}(\varphi)$, $\varphi \Rightarrow \mathcal{P}(\varphi)$ and if $\mathcal{I}(\varphi_1, \varphi_2) = True$ then $\varphi_1 \Rightarrow \varphi_2$.*
b. *The constraints φ_r that represent properties of built-in predicates r are sound.*

Requirement 2.2 (Termination)
a. Monotonicity*: There is a partial order $\sqsubseteq$ such that $\mathcal{S}$ and $\mathcal{P}$ are monotonic w.r.t. $\sqsubseteq$, and $\mathcal{I}$ is complete w.r.t. $\sqsubseteq$.*
b. Finite Ascending Chains*: The range of $\mathcal{P}$ has no infinite strictly ascending chain w.r.t. $\sqsubseteq$. That is, for every (possibly infinite) chain $\varphi_1 \sqsubseteq \varphi_2 \sqsubseteq \cdots$, the set $\{\mathcal{P}(\varphi_1), \mathcal{P}(\varphi_2), \ldots\}$ is finite.*

[2]Note that these equations correspond to Jacobi's method of successive approximations (see [9], page 136).

Theorem 2.1 (Soundness) *The fix-point iteration procedure correctly computes $p^{\#}$ for recursive predicates p.*

Theorem 2.2 (Termination) *For every predicate p in the program, $p^{\#}$ can be effectively computed.*

3 Boolean Simplification Layer

Given that conjunctions and disjunctions are the primary operations used by the generic layer, an obvious representation for program properties is to use a disjunctive normal form, where each component of a conjunction is an atomic constraint. (The structure of the atomic constraints themselves is defined by the subsequent layer.) A DNF is capable of capturing simple case-splitting that occurs in programs due to the presence of multiple clauses defining a single predicate. Although DNF's can become very large in the worst case, this does not seem to happen in practice[3]. The canonical form chosen by this layer is thus a DNF representation.

In order to provide the *simplify* and *implication* operations on DNF's, this layer makes use of the *quotient* and *product* operators supplied by the atomic constraints layer. Note that it is in the atomic constraints layer that these operations are defined and the notion of canonical form is fixed. However, we describe the properties of these operations here in order to explain the operations of the boolean simplification layer.

The *product* of two atomic constraints C_1 and C_2 is a conjunction $(C_1' \wedge \cdots \wedge C_k')$ that approximates $(C_1 \wedge C_2)$. This operation defines the notion of *orthogonality*: two atomic constraints C_1 and C_2 are orthogonal iff $C_1 * C_2 = C_1 \wedge C_2$. A conjunction $(C_1 \wedge \cdots \wedge C_n)$ is said to be in *canonical form* iff the C_i's are pairwise orthogonal. Observe that the conjunction $(X = a) \wedge (Y = a)$ can be represented (without redundant information) in several alternate ways, such as $(X = Y) \wedge (Y = a)$, $(X = Y) \wedge (X = a)$, and so on. The product operation is used to choose *one* of the possible representations, say, $(X = Y) \wedge (Y = a)$. With this choice of product, $X = a$ and $Y = a$ are *not* orthogonal, whereas $X = Y$ and $Y = a$ are orthogonal. The properties of the product operation are formalized in requirement 3.1 below.

The *quotient* operation is defined as follows. Intuitively, given that C_2 is satisfied, the quotient C_1/C_2 gives the extra conditions that are needed to satisfy C_1. For example, given $X = c(Y)$ and we want to prove that $X = c(b)$; then the additional constraint we need is $Y = b$. Hence $(X = c(b))/(X = c(Y)) = (Y = b)$. In the definition of quotient, we exploit the fact that it is used only in the context of formulas already in canonical form. For instance, given $X = t$, the additional constraints we need to have for showing $Y = t$ is one of $\{Y = t, X = Y, Y = X\}$, *i.e.*, a disjunction of these three constraints. However, disjunctions are not permitted in the above definition of quotient. We have chosen our canonical form so that only one of these three constraints can be in a conjunction that contains $X = t$, and are hence able to define the quotient operation.

Based on these functions, the realization of *simplify*, *implication* and *project* is sketched in figure 2. In this figure, we use φ to denote a DNF, ψ to denote a conjunct and C (with or without subscripts or primes) to denote atomic constraints. The algorithms in the figure are expressed using oriented equations, from which a computational procedure can be readily derived using the following "meta" rules: (1) the rules are matched by treating $\wedge$ and $\vee$ as associative-commutative symbols (*e.g.*, when $C \wedge \psi$ matches $C_1 \wedge C_2 \wedge C_3$, C can match any of C_1, C_2, or C_3), and (2) a

[3] Our experimental results indicate that the average number of conjunctions that arise in a DNF is very close to 1, and rarely exceeds 1.2.

$$
\begin{array}{llcl}
1. & \mathcal{S}_\vee(\varphi_1, \varphi_2) & \rightarrow & \mathit{absorb}(\varphi_1 \vee \varphi_2) \\
2. & \mathcal{S}_\wedge(\varphi_1, \varphi_2) & \rightarrow & \mathit{absorb}(\prod(\varphi_1, \varphi_2)) \\
3a. & \mathcal{I}(\psi_1 \vee \varphi_1, \varphi_2) & \rightarrow & [\exists \psi_2 \in \varphi_2 \; (\mathit{conj\_quotient}(\psi_2, \psi_1) = \mathit{True})] \wedge \mathcal{I}(\varphi_1, \varphi_2) \\
3b. & \mathcal{I}(\mathit{False}, \varphi_2) & \rightarrow & \mathit{True} \\
4. & \mathcal{P}^{\overline{X}}(\psi_1 \vee \cdots \vee \psi_n) & \rightarrow & \mathit{absorb}(\mathcal{P}_\wedge{}^{\overline{X}}(\psi_1) \vee \cdots \vee \mathcal{P}_\wedge{}^{\overline{X}}(\psi_n)) \\
5a. & \mathit{absorb}(\psi_1 \vee \psi_2 \vee \varphi) & \rightarrow & \text{if } (\mathit{conj\_quotient}(\psi_2, \psi_1) = \mathit{True}) \text{ then } \mathit{absorb}(\psi_2 \vee \varphi) \\
5b. & \mathit{absorb}(\varphi) & \rightarrow & \varphi \\
6a. & \mathit{conj\_quotient}(C \wedge \psi_1, \psi_2) & \rightarrow & \mathit{cons\_quotient}(C, \psi_2) \wedge \mathit{conj\_quotient}(\psi_1, \psi_2) \\
6b. & \mathit{conj\_quotient}(\mathit{True}, \psi_2) & \rightarrow & \mathit{True} \\
7a. & \mathit{cons\_quotient}(C, C_1 \wedge \psi) & \rightarrow & \text{if } (C/C_1 = \mathit{True}) \text{ then } \mathit{True} \\
 & & & \quad \text{else} \quad \text{if } (C/C_1 \neq C) \text{ then} \\
 & & & \qquad\qquad \mathit{conj\_quotient}(C/C_1, C_1 \wedge \psi) \\
7b. & \mathit{cons\_quotient}(C, \psi) & \rightarrow & C \\
8. & \prod((\psi_1 \vee \cdots \vee \psi_n), (\psi'_1 \vee \cdots \vee \psi'_m)) & \rightarrow & \bigvee_{i,j} \mathit{conj\_product}(\psi_i, \psi'_j), \quad (1 \leq i \leq n, 1 \leq j \leq m) \\
9a. & \mathit{conj\_product}(\psi_1, C \wedge \psi_2) & \rightarrow & \mathit{cons\_product}(\mathit{conj\_product}(\psi_1, \psi_2), C) \\
9b. & \mathit{conj\_product}(\psi_1, \mathit{True}) & \rightarrow & \psi_1 \\
10a. & \mathit{cons\_product}(C_1 \wedge \psi, C) & \rightarrow & \text{if } (C/C_1 = \mathit{True}) \text{ then } C_1 \wedge \psi \\
 & & & \quad \text{else} \quad \text{if } (C/C_1 = C) \text{ then} \\
 & & & \qquad\qquad \text{if } (C_1/C = C_1) \text{ then} \\
 & & & \qquad\qquad\quad C_1 \wedge \mathit{cons\_product}(\psi, C) \\
 & & & \qquad\qquad \text{else} \quad \mathit{conj\_product}(\mathit{cons\_product}(\psi, C), C_1/C) \\
 & & & \quad \text{else} \quad \mathit{conj\_product}(\psi, C * C_1) \\
10b. & \mathit{cons\_product}(\mathit{True}, C) & \rightarrow & C
\end{array}
$$

Figure 2: Equations defining $\mathcal{S}$, $\mathcal{P}$ and $\mathcal{I}$.

rule that appears later in the text is used only when earlier rules are not applicable. For example, $\mathit{absorb}(\varphi)$ can be implemented as follows: if there are two conjunctions ψ_1, ψ_2 in φ such that $\mathit{conj\_quotient}(\psi_2, \psi_1) = \mathit{True}$, then invoke $\mathit{absorb}(\varphi - \phi_1)$, *i.e.*, eliminate ψ_1 from φ; otherwise, return φ.

The simplification function uses absorption laws to eliminate redundant conjunctions, *i.e.*, conjunctions that are implied by other conjunctions in the same DNF. Both the simplification function, as well as the implication function, make use of *conj_quotient* to determine if one conjunction implies another. Actually, *conj_quotient* is an extension of the quotient operation on conjunctions, and hence is a generalization of the implication operation on conjunctions. It uses *cons_quotient* in turn, which is ultimately realized in terms of the quotient operation provided by the atomic constraints layer. The functions *conj_product* and *cons_product* extend the product operation in the same way *conj_quotient* and *cons_quotient* extend the quotient operation. The projection function $\mathcal{P}$ is performed by projecting each conjunction using $\mathcal{P}_\wedge$ followed by the *absorb* operation to remove redundant conjunctions from the result.

Proof Obligations The soundness and termination of this layer are established based on the following properties of $*$, $/$ and $\mathcal{P}_\wedge$.

Requirement 3.1 *The product $(C_1 * C_2)$ of two atomic constraints C_1 and C_2 is a conjunction $C_1' \wedge \cdots \wedge C_k'$ in canonical form such that $(C_1 \wedge C_2) \Rightarrow (C_1' \wedge \cdots \wedge C_k')$.*

Requirement 3.2 *The quotient C_1/C_2 of two atomic constraints C_1 and C_2 is a conjunction $C_1' \wedge \cdots \wedge C_k'$ in canonical form such that $\forall \psi$ such that $(C_2 \wedge \psi)$ is in canonical form, $[\psi \Rightarrow (C_1' \wedge \cdots \wedge C_k')] \Leftrightarrow [(C_2 \wedge \psi) \Rightarrow C_1]$.*

Requirement 3.3 *$*$ is monotonic; furthermore, $\forall C \in C_1 * C_2$, $C <_{wfo} C_1$ or $C <_{wfo} C_2$ or $C_1 * C_2 = C1$ or $C_1 * C_2 = C_2$, where $<_{wfo}$ is some well-founded order.*

Requirement 3.4 *$\forall C \in C_1/C_2$, $C <_{wfo} C_1$ or $C_1/C_2 = C_1$.*

Requirement 3.5 *$\mathcal{P}_\wedge$ is sound, monotonic and the range of $\mathcal{P}_\wedge$ has no infinite strictly ascending chain.*

Theorem 3.1 (Soundness) *$\mathcal{S}$, $\mathcal{P}$ and $\mathcal{I}$ are sound.*

Theorem 3.2 (Termination) *There exists a partial order $\sqsubseteq$ such that $\mathcal{I}$ is complete, $\mathcal{S}$ and $\mathcal{P}$ are monotonic, and range of $\mathcal{P}$ has no infinite ascending chain w.r.t. $\sqsubseteq$.*

4 Atomic Constraint Layer

This layer deals with atomic constraints and provides the operations $/$, $*$ and $\mathcal{P}_\wedge$ and operations needed by the boolean formula layer. We first need to select the type of atomic constraints supported and specify their canonical form. In this paper, we have chosen the *equality* and *membership* atomic constraints. Analyses that can be modeled using these two constraints include, for example, those that are typically formulated using *Prop* domain (proposed by Marriot and Sondergaard in [18] and refined by Cortesi et al [7]). The equality constraint enables us to capture aliasing, whereas membership constraints enable us to capture the notion of a variable taking a substitution from an abstract domain. The membership constraint views each domain point as representing a subset of the Herbrand base of a program (commonly referred to as the *concretization* of the domain point), and thus membership implies that a variable takes its substitution from this set. For instance, in groundness analysis, if g is a point in the abstract domain representing the set of all ground terms, an abstract substitution of the form $[X \leftarrow g]$ is equivalent to the constraint $X \in g$. Similarly any abstract domain can be described by a domain $\mathcal{D}$ of subsets of the universe of terms.

More concretely, the (canonical representation of) constraints are of the form $X = Y$ or $X = c(Y_1, \ldots, Y_n)$ or $X \in d$ or $X \in c(d_1, \ldots, d_n)$, where c is a function symbol, $X, Y, Y_1, \ldots, Y_n$ are variables and $d, d_1, \ldots, d_n$ are elements of the abstract domain. We refer to any one of these constraints an atomic constraint *on* X. Observe that although this representation restricts the rhs of the constraints to be of depth 1, term structure constraints of any depth can be captured using intermediate variables, e.g., $X = c(t_1, \ldots, t_n)$ can be captured as the conjunction $(X = c(Y_1, \ldots, Y_n)) \wedge (Y_1 = t_1) \wedge \cdots \wedge (Y_n = t_n)$. We prevent duplicate representation of the equality constraint between variables (*i.e.*, $X = Y$ which can also be represented as $Y = X$) by defining a total order on the set of variables and ensuring that if $X = Y$ occurs as a constraint then X precedes Y in that order. Given orthogonality, this condition actually ensures that any conjunction of atomic constraints that is in canonical form will contain at most one constraint on any one variable. This fact, together with the ordering, enables us to identify all the constraints on a variable X by recursively following the constraints on variables appearing on the rhs of the constraint on X in an efficient manner. This fact is exploited in our definition of the $\mathcal{P}_\wedge$ function.

$$
\begin{array}{lrcl}
0. & C/C & \rightarrow & \mathit{True} \\
1a. & (X=t)/(Y=t) & \rightarrow & (Y=X) \quad \text{if } Y>X \\
 & & & (X=t) \quad \text{otherwise} \\
1b. & C(X,t)/C'(X',t') & \rightarrow & C(X,t), \quad \text{whenever } X \neq X' \\
2a. & (X=t)/(X\in t) & \rightarrow & \mathit{True} \\
2b. & (X=t)/(X\in t') & \rightarrow & (X=t) \\
3a. & (X\in t)/(X=Y) & \rightarrow & (Y\in t) \\
3b. & (X\in d)/(X=c(Y_1,...,Y_n)) & \rightarrow & \bigwedge_{i=1}^{n}(Y_i\in d_i) \quad \text{where } d_i = \mathit{subset}(d,c,i) \\
3c. & (X\in c(...))/(X=c'(...)) & \rightarrow & \mathit{False}, \quad \text{if } c\neq c' \\
3d. & (X\in c(d_1,...,d_n))/(X=c(Y_1,...,Y_n)) & \rightarrow & \bigwedge_{i=1}^{n}(Y_i\in d_i) \\
4a. & (X=Y)/(X=t) & \rightarrow & (Y=t) \\
4b. & (X=t)/(X=Z) & \rightarrow & (Z=t) \\
4c. & (X=c(...))/(X=c'(...)) & \rightarrow & \mathit{False}, \quad \text{if } c\neq c' \\
4d. & (X=c(Y_1,...,Y_n))/(X=c(Y_1',...,Y_n')) & \rightarrow & \bigwedge_{i=1}^{n}(Y_i'=Y_i) \\
5a. & (X\in d_1)/(X\in d_2) & \rightarrow & \text{if } (\mathit{glb}(d_1,d_2)=d_2) \text{ then } \mathit{True} \\
 & & & \text{else } (X\in d_1) \\
5b. & (X\in c(d_1,...,d_n))/(X\in d') & \rightarrow & (X\in c(d_1,...,d_n)) \\
5c. & (X\in c(...))/(X\in c'(...)) & \rightarrow & \mathit{False}, \quad \text{if } c\neq c' \\
5d. & (X\in c(d_1,...,d_n))/(X\in c(d_1',...,d_n')) & \rightarrow & \text{if } (\mathit{glb}(d_i,d_i')=d_i') \text{ then } \mathit{True} \\
 & & & \text{else } (X\in c(d_1,...,d_n))
\end{array}
$$

Figure 3: Equations defining quotient (/).

Having defined the canonical form of atomic constraints, we now describe the operations needed by this layer from the domain layer. The *quotient* and *product* operations are defined using two domain-dependent functions, called *glb* and *subset*. The function *glb* computes the greatest lower bound of two domain elements, which corresponds to (an approximation of) the intersection of the sets of terms represented by the two domain points. For description of *subset* let d be a domain constant, c be a function symbol and i an integer. The value returned by *subset(d,c,i)* is the smallest domain element d' such that $c(t_1,\ldots,t_n) \in d \Rightarrow t_i \in d'$. For example, in order for $c(Y_1,\ldots,Y_n)$ to be in g we need $Y_1 \in g,\ldots,Y_n \in g$. Hence $\mathit{subset}(g,c,i) = g$ for $1 \le i \le n$. The $\mathcal{P}_\wedge$ function requires a third operation from the domain layer, named *superset*, which takes an argument t of the form $c(d_1,\ldots,d_n)$ and returns a domain element that is the minimal superset of its argument t. More formally, let S be the set of terms that are instances[4] of t. Then $\mathit{superset}(t)$ returns the smallest domain element that corresponds to a superset of S. Properties of the domain-dependent functions are formalized in requirements 4.1, 4.2 and 4.3.

The procedure for computing quotient is specified by the rules in Figure 3. Product operation is defined along the same lines (see appendix, section C). In the figure, d (with or without subscripts) denotes some point in the domain. Equation 2b assumes that equality constraints are never partially implied by $\in$ constraints and equation 5b assumes that not all terms contained in a domain element have the same root symbol. The proofs of soundness and termination of / and $*$ can be easily obtained, based on the properties of *glb* and *subset.*

We now sketch the definition of the projection function $\mathcal{P}_\wedge$ on conjuncts. We ensure that $\mathcal{P}_\wedge$ has a finite range by restricting the *depth* of the projected constraints to some constant k, similar to the

[4] An instance of $c(d_1,\ldots,d_n)$ is a term $c(s_1,\ldots,s_n)$ such that $\forall i \quad s_i \in d_i$.

depth-k abstraction of Sato and Tamaki [22]. The depth-k approximations of a set of constraints is computed by replacing the constraints on variables that occur below depth k by their *zero-depth approximations.* The zero-depth approximation of a constraint $C(X,t)$ is found by replacing t with its zero-depth approximation. Zero-depth approximations of variable-free terms are given by a domain dependent function *superset*. This procedure is illustrated by the following example.

Consider the conjunct $\psi = (X = c(Y)) \wedge (Y = b(Z)) \wedge (Z \in g)$. To perform a depth-1 projection of ψ onto $\{X\}$ we first find zero-depth approximation of the constraint on Y, which in turn will need zero-depth approximations of the constraints on Z. The constraint on Z is already its zero-depth approximation. Assuming $superset(b(g)) = g$, we obtain the zero depth approximation for Y as $Y \in g$. Substituting this into constraint on X gives the depth-1 projection of ψ on X as $(X \in c(g))$.

After propagating and truncating constraints as described above, intermediate variables that occur in depth-k constraints are renamed consistently. In particular, variables that represent the same position in a term on the rhs of a constraint on a specific variable have the same name. This is done to ensure that implication operation need not check for consistency of the constraints on the intermediate variables. The proofs of soundness of $\mathcal{P}_\wedge$ is routine and is based on the following correctness requirements of the domain layer functions.

Requirement 4.1 *(glb):* $\forall d_1, d_2 \quad \forall t \quad t \in d_1 \wedge t \in d_2 \Rightarrow t \in glb(d_1, d_2)$.

Requirement 4.2 *(subset):* $\forall d' \quad (c(t_1, \ldots, t_n) \in d \Rightarrow t_i \in d') \Rightarrow subset(c, d, i) \subseteq d'$.

Requirement 4.3 *(superset):* $\forall d \quad (c(t_1, \ldots t_n) \in d \wedge \forall i \quad t_i \in d_i) \Rightarrow superset(d_1, \ldots, d_n) \subseteq d$.

Requirement 4.4 *The domain $\mathcal{D}$ has no infinite strictly ascending chain.*

5 Domain Layer

We illustrate the domain dependent functions by formulating type analysis in our framework, with the the domain $\mathcal{D} = \{\top, g, list, \bot\}$ where $\top$ represents the universe of terms, $\bot$ represents the empty set, *list* denotes lists and g denotes non-list ground terms. *glb* is defined such that $\top$ is the greatest element, $\bot$ is the least and *list* and g are incomparable; *i.e.*, $glb(list, g) = glb(g, list) = \bot$. The functions *superset* and *subset* are defined by the following equations (here d denotes some arbitrary element in $\mathcal{D}$ and *cons* represents the list constructor):

$$\begin{aligned} subset(list, cons, 1) &= \top \\ subset(list, cons, 2) &= list \\ subset(d, c, i) &= d, \quad \forall i, \quad d \neq list \\ superset(d) &= d \\ superset([d_1|d_2]) &= list \quad \text{if } d_2 = list, \quad \top \text{ otherwise} \\ superset([\,]) &= list \\ superset(c(t_1, \ldots, t_n)) &= g \quad \text{if } \forall i \; t_i = g, \quad \top \text{ otherwise} \end{aligned}$$

The base constraints for this analysis are provided for the built-in predicates such as $<$, *is*, etc. For example, $is^{\#}(X,Y) = (X = Y) \wedge (Y \in g)$. The soundness proofs for *glb*, *subset*, *superset* and the base constraints are straightforward and omitted.

As an example of the type analysis as formulated here, consider the *append* predicate:

$$\begin{aligned} &append([\,], Y, Y). \\ &append([U|V], Y, [U|W]) \quad :- \quad append(V, Y, W). \end{aligned}$$

The abstract equation defining $append^{\#}$ is:

$$\begin{aligned} append^{\#}(X,Y,Z) \quad = \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_{\vee}(\quad & \mathcal{S}_{\wedge}(X = [\,] \wedge Y = Z) \quad \vee \\ & \mathcal{S}_{\wedge}(X = [U|V] \wedge Z = [U|W] \wedge append^{\#}(V,Y,W)))) \end{aligned}$$

We obtain the following iterations. (See appendix for a more detailed illustration of the example.)

$$\begin{aligned} append^{\#,0} &= False \\ append^{\#,1} &= (X = [\,]) \wedge (Y = Z) \\ append^{\#,2} &= (X \in list) \\ append^{\#,3} &= (X \in list) \end{aligned}$$

Note that fix point is reached after 2 iterations. Thus, we infer that X is always a list, but nothing is known about Y and Z, since $append([\,], t, t)$ succeeds for any term t.

6 Implementation

Based on our framework, we implemented groundness and type analyses. All analyses reported in this section share the generic, boolean simplification and atomic constraint layers. The domain dependent functions are specified separately for each analysis. These functions form only 10% of the code, thereby allowing us to reuse the other 90% across all the analyses.

In Table 1, we compare the performance of groundness analysis in our method (listed under 'SCS', short for Symbolic Constraint Solving) with that on the GAIA with re-execution [16]. The performance of both the implementations were evaluated on a number of programs. The programs and input modes used were the same as those used in [16]. The programs are: an alpha-beta procedure `Kalah`, an equation solver `Press`, a cutting-stock program `CS`, a disjunctive scheduling program `Disj`, a peephole optimizer `Peep`, a planning program `Plan`, an n-queens program `Queens`, a tokenizer `Read` by R. O'Keefe and D.H.D. Warren, a program `PG` by W. Older and a quicksort program `Qsort`.

The accuracies of the implementations are listed in the table as the percentage of variables determined to be ground by each implementation. The table shows that SCS achieves essentially the same accuracies reported by GAIA. The GAIA timings were taken from [16] and the timings for SCS were obtained on a Sparcstation-1, the same architecture as used in [16]. In order to compute both input and output modes (as done in GAIA), SCS analyses were performed after Magic Templates transformation [21]. The execution times for SCS given in the table *include the transformation time.* The table shows that the SCS is three to four times faster than the GAIA. It should be noted, however, whereas the SCS figures are for groundness analysis, GAIA figures are for extracting both groundness and freeness together.

In Table 2, we compare the performance of our method with that of the implementation based on Toupie system [5], on the following domains:

- `2pt-ground`: Two point domain for groundness,
- `3pt-ground`: Three point domain for groundness,
- `4pt-types`: Four point domain ({`int, const, list, funct`}) for type,
- `5pt-types`: Five point domain {`int, const, list, funct, u`} for type, and
- `BigType`: The type domain in figure 4.

The first four domains correspond respectively to the `Prop`, `Prop+`, `Types` and `Types+` domains used in [5]. The timings for Toupie are not available for `BigType`. The timings for SCS were

	Accuracy (% vars ground)		Execution Time (seconds)	
	GAIA	SCS	GAIA	SCS
`CS`	100	100	4.32	1.09
`Disj`	100	100	2.42	0.63
`Kalah`	98	98	1.88	0.55
`Peep`	86	87	2.92	0.70
`PG`	100	100	0.27	0.10
`Plan`	97	97	0.21	0.09
`Press`	27	27	7.73	1.37
`Qsort`	78	78	0.11	0.05
`Queens`	100	100	0.07	0.02
`Read`	57	62	4.37	1.09

Table 1: Performance of SCS compared to GAIA.

	`2pt-ground`		`3pt-ground`		`4pt-types`		`5pt-types`		`BigType`
	Toupie	SCS	Toupie	SCS	Toupie	SCS	Toupie	SCS	SCS
`CS`	0.53	0.14	0.90	0.18	0.20	0.12	0.21	0.12	0.14
`Disj`	0.48	0.17	0.68	0.22	0.30	0.12	0.36	0.13	0.15
`Gabriel`	0.18	0.04	0.26	0.05	0.18	0.04	0.21	0.04	0.05
`Kalah`	0.50	0.11	0.80	0.15	0.35	0.12	0.43	0.12	0.13
`Peep`	0.83	0.08	1.30	0.11	0.55	0.10	0.70	0.10	0.11
`PG`	0.11	0.02	0.16	0.03	0.11	0.02	0.15	0.02	0.03
`Plan`	0.10	0.02	0.15	0.02	0.06	0.02	0.11	0.02	0.03
`Press`	0.76	0.15	1.20	0.20	0.68	0.16	1.00	0.17	0.17
`Qsort`	0.03	0.01	0.05	0.01	0.06	0.01	0.08	0.01	0.01
`Queens`	0.03	0.01	0.06	0.01	0.03	0.01	0.03	0.01	0.01
`Read`	0.83	0.25	1.51	0.28	2.40	0.26	4.05	0.26	0.27

Table 2: Performance compared to Toupie (*all times in seconds*).

obtained on a Sparc IPX, the same architecture as used in [5]. The results show that SCS is on the average about 5 times faster than the Toupie system. The differences between the accuracies of these two implementations are also negligible. However the results reflect information obtained independent of the calling context and greater accuracy can be achieved by post-processing for a given context. Due to the differences in the representations, the results of post-processing may differ for the two methods, as discussed in Section 7.

Dependency on Domain Size Note that when domain size is increased from 2 to 3 (**`2pt-ground`** to **`3pt-ground`**) the analysis times went up by 30% on the average on SCS, compared to 60% on the Toupie system. Increasing the domain size from 4 to 5 (**`4pt-types`** to **`5pt-types`**) results in less then 2% increase in analysis times on SCS whereas the analysis times on the Toupie system increase by 50% on the average (and 30% if extreme data points are omitted).

We performed type analysis on a 14-point domain (figure 4) in order to measure the domain sensitivity of our approach. In figure 4, '$[\alpha]$' denotes a list of type α and '$\top$' denotes unknown

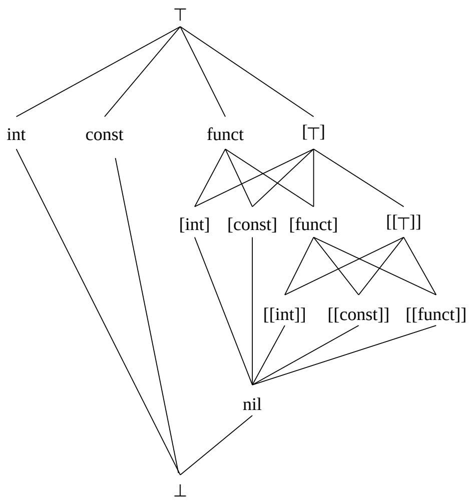

Figure 4: Structure of the 14-point type domain (**BigType**)

```
q(X, Y) :- p1(a, X, Y).
p1(Z, X, [Z|Y]).
```

(a)

```
r(Y) :- p2(X, X, Y).
p2(a, b, Z).
p2(X, X, []).
```

(b)

Figure 5: Programs illustrating differences in accuracy.

type. The analysis times on this domain are given in the last column of table 2. Note that when the domain size is increased from 5 to 14, (an increase of 280%) the analysis time increases, on the average, by only 10%. This shows that the SCS framework is largely insensitive to domain size.

7 Discussion

In our method, the constraints are interpreted over the concrete domain and abstraction itself is done only when lossy constraint solving operations are performed. Thus, our method can be considered as a bottom-up abstract interpretation technique over the concrete domain. The use of lossy operations on relations in our framework corresponds to the use of lossy conjunction and disjunction of Wegbreit [23] and the lossy composition of Codognet and File [4] and to the notion of *insertion* due to Marriot and Sondergaard [19].

In contrast to our use of lossy operations, the loss of information in a *Prop*-domain based system such as [5] occurs only at the abstraction step, and the resulting constraints are solved exactly. It [10] Cousot and Cousot showed that loss of information at iteration time can yield more precise analyses compared to the Galois-connection based approaches where loss of information occurs at abstraction time. However, since the language used to represent the properties in our framework and *Prop*-domain based systems differs considerably, the accuracy of the two approaches are incomparable. For instance, consider groundness analysis of predicate q in Figure 5a. Using membership and equality constraints in our method as described in Section 4 yields no information. However, analyses using the *Prop* domain can conclude that $X \in g$ iff $Y \in g$. On the other hand, groundness analysis of predicate r in Figure 5b using the *Prop* domain does not yield any information since a and b are mapped to the same point in the abstract domain. However, in our framework, since constraints are interpreted over the concrete domain, we can conclude that $Y \in g$. Although the accuracies of the two approaches can potentially differ, experiments indicate that our method and those in [5, 16] have similar accuracies in practice.

The accuracy of analyses in our framework can be improved in several ways. Firstly, the operations in atomic constraint layer can be extended to return disjunctions instead of conjunctions and the rules of the boolean simplification layer can be suitably modified. Secondly, additional constraints, such as those that represent sharing and containment relations, can be introduced by modifying the atomic constraints layer[5]. Note that both approaches enlarge the set of representable properties, thereby increasing accuracy since the loss of information is governed by what can be represented. With either of these extensions, groundness analysis on the example in Figure 5a yields $(X \in g, Y \in g) \vee (X \in ng, Y \in ng)$.

8 Conclusions and Future Work

We presented a modular framework that implements bottom-up abstract interpretation of logic programs using symbolic constraint solving techniques. Such a framework needs to balance the conflicting requirements of providing efficient constraint solving algorithms for particular types of constraints and at the same time being general enough to deal with a large class of constraints. The framework presented here attempts to satisfy these requirements by dividing the analysis and constraint solving operations into many layers, thereby sharing these operations across different kinds of constraints. Each layer provides a set of operations for the higher layer using the operations provided by the lower layers. Computations within each layer are independent of the implementation as well as the constraint representation used in the other layers. Interfaces between the layers are characterized by a set of operations and corresponding correctness requirements. The correctness of the complete framework is established based on these interface requirements. A prototype implementation of our framework shows that it scales very well to large domains and furthermore, compares favorably with existing implementations of other analysis methods.

The framework can be easily extended to domains without ascending chain condition, by introducing widening operators [12]. In particular, the implication operation, instead of returning true or false, can be modified to invoke the widening operator if the implication is false, thus returning the next (extrapolated) iterate. Note that the finiteness condition on the project operation in atomic constraints layer (requirement 3.5) can now be relaxed, leading to the relaxation of the finiteness condition on the domain (requirement 4.4) in the domain layer. Modularizing the widening operations along the same lines as *simplify* and *project* while preserving efficiency is a topic of current research.

Other possible extensions to the framework can be based on modifying the requirements on the different operators: by weakening them, thereby yielding more general analyses or strengthening them to make the analyses more efficient. For example, the completeness condition on the quotient operator in the atomic constraints layer can be weakened (analogous to the weakening of completeness of implication in boolean formula layer), thus enabling quotients to be defined for a larger class of constraints. Further research is needed to investigate the tradeoff between relaxing and tightening the requirements.

[5] As an illustration, in section D of Appendix, we present the rules added to the atomic constraints layer for handling constraints that represent sharing.

Acknowledgements

We thank Pascal Van Hentenryck for providing us with the benchmark programs, and Marc-Michel Corsini, Saumya Debray, Pascal Van Hentenryck and David S. Warren for their comments on an earlier version of this paper.

References

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

[2] R.E. Bryant. Ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[3] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a Magic wand. In *International Logic Programming Symposium*, pages 114–129. MIT Press, 1993.

[4] P. Codognet and G. File. Computations, abstractions and constraints. In *International Conference on Computer Languages*, pages 155–164. IEEE Press, 1992.

[5] M-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In *International Symposium on Programming Language Implementation and Logic Programming*, number 714 in LNCS, pages 75–91. Springer Verlag, 1993.

[6] A. Cortesi and G. File. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 52–61. ACM Press, 1991.

[7] A. Cortesi, G. File, and W. Winsborough. Prop revisited: Propositional formula as abstract domain for groundness analysis. In *IEEE Symbosium on Logic in Computer Science*, pages 322–327. IEEE Press, 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Priciples of Programming Languages*, pages 238–252. ACM Press, 1977.

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[10] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, number 631 in LNCS, pages 269–295. Springer Verlag, 1992.

[11] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to Comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages. In *International Conference on Computer Languages*, pages 95–112. IEEE Press, 1994.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a progam. In *ACM Symposium on Priciples of Programming Languages*, pages 84–96. ACM Press, 1978.

[13] R. Giacobazzi, S. Debray, and G. Levy. A generalized semantics for constraint logic programs. In *International Conference on Fifth Generation Computing Systems*, 1992.

[14] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In *North American Conference on Logic Programming*, pages 154–165. MIT Press, 1989.

[15] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, pages 503–582, 10th Anniversary Special Issue 1994.

[16] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. In *Joint International Conference/Symposium on Logic Programming*, pages 750–764. MIT Press, 1992.

[17] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, January 1994.

[18] K. Marriot and H. Sondergaard. Notes for a tutorial on abstract interpretation of logic programs (unpublished). In *North American Conference on Logic Programming*, 1989.

[19] K. Marriot and H. Sondergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13:181–204, 1992.

[20] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13:315–347, 1992.

[21] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programming. In *Joint International Conference/Symposium on Logic Programming*, pages 140–159. MIT Press, 1988.

[22] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[23] B. Wegbreit. Property extraction in well founded property sets. *IEEE Transactions on Software Engineering*, SE-1(3):270–285, September 1975.

Appendix

(This appendix contains only supplementary reference material.)

A Proofs for the Generic Layer

A.1 Soundness

We first consider only non-recursive programs and use this result to prove soundness of the general case.

Lemma A.1 *Let p be a predicate as defined before (Equation 1 on page 3), and $(q_k^j)^{\#}$ be sound for each q_k^j respectively. Then $p^{\#} = \mathcal{S}_{\vee}(\bigvee_{j=1}^{n} \mathcal{P}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} (q_k^j)^{\#})))$ is sound for p.*

Proof: *(Sketch)* Given that ${s_1}^{\#}(\overline{Y})$ and ${s_2}^{\#}(\overline{Z})$ are necessary conditions for $s_1(\overline{Y})$ and $s_2(\overline{Z})$ respectively, then clearly ${s_1}^{\#}(\overline{Y}) \wedge {s_2}^{\#}(\overline{Z})$ is a necessary condition for the goal $s_1(\overline{Y}), s_2(\overline{Z})$. It is easy to see that $\bigwedge_{k=1}^{m_j} {q_k^j}^{\#}(\overline{X_k^j})$ is a necessary condition for the goal $q_1^j(\overline{X_1^j}), \ldots, q_{m_j}^j(\overline{X_{r_j}^j})$. From requirement 2.1a we have $\phi \Rightarrow \mathcal{P}(\mathcal{S}_{\wedge}(\phi))$, and it follows that $\mathcal{P}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} {q_k^j}^{\#}(\overline{X_k^j})))$ is a necessary condition for the j^{th} clause $p_j(\overline{X})$.

Furthermore, ${s_1}^{\#}(\overline{Y}) \vee {s_2}^{\#}(\overline{Z})$ is a necessary condition for the goal $s_1(\overline{Y}); s_2(\overline{Z})$ whenever ${s_1}^{\#}(\overline{Y})$ and ${s_2}^{\#}(\overline{Z})$ are necessary conditions for $s_1(\overline{Y})$ and $s_2(\overline{Z})$ respectively. It follows that $\mathcal{S}_{\vee}(\bigvee_{j=1}^{n} \mathcal{P}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} (q_k^j)^{\#})))$ is a necessary condition for predicate p. ■

Theorem A.2 *For each predicate p in a nonrecursive program $p^{\#}$ is a necessary condition for p.*

Proof: By induction on height of the dependency graph (dag). The base case relates the predicates that are at the bottom of the call graph – i.e., those that are pre-defined or *built-in*. This is a direct consequence of requirement 2.1b. The rest of the proof follows easily from lemma A.1. ■

The above theorem is used to establish that every fix point of p is a necessary condition.

Theorem A.3 *If φ_p is a fix point for p, then φ_p is a necessary condition for p.*

Proof: Let $\varphi_p, \varphi_q, \ldots$ be the fix points for $p, q, \ldots$. For each predicate p, define $p^0, p^1, \ldots$ as follows. For each rule that defines p of the form

$$p(\overline{X}) :\!-\ Body.$$

define

$$p^0(\overline{X}) :\!-\ \text{loop}.$$

and for $i > 0$

$$p^i(\overline{X}) :\!-\ \Gamma_i(Body).$$

where loop represents some non-terminating computation $\Gamma_i(s)$ replaces each occurrence of any predicate symbol q in s by q^i.

Since p^0 never terminates, φ_p is a safe necessary condition for p^0. Note that p^i is nonrecursive for $i > 0$. By taking the necessary condition for p^0 as φ_p, we get the necessary condition for p^1 as φ_p, since φ_p is a fix point. By induction we obtain φ_p as a necessary condition for p^i, for all i.

The behavior of p and p^i is identical for all execution sequences of length $\leq i$. Given any execution sequence of p, say of length n, behavior of p is identical to that of p^n; and φ_p is a necessary condition for p^n. Hence φ_p is a necessary condition for p. ■

The soundness theorem 2.1 follows as a corollary to the above theorem.

A.2 Termination

Lemma A.4 *For every predicate p in the program, $p^{\#,i} \sqsubseteq p^{\#,i+1}$.*

Proof: The proof is by induction on i. For the base case, $p^{\#,0} \sqsubseteq p^{\#,1}$ for every predicate p. For nonrecursive predicates, $p^{\#,0} = p^{\#} = p^{\#,1}$, and hence $p^{\#,0} \sqsubseteq p^{\#,1}$. For recursive predicates $p^{\#,0} = \mathit{False}$ and $\mathit{False} \sqsubseteq \phi$ for every $\phi \in \Phi$. Hence $p^{\#,0} \sqsubseteq p^{\#,1}$.

For the induction step, assume that $p^{\#,i} \sqsubseteq p^{\#,i+1}$ for all $i < m$. Let ψ_m denote the formula $\mathcal{S}_\vee(\bigvee_{j=1}^{n} \mathcal{P}(\mathcal{S}_\wedge(\bigwedge_{k=1}^{r_j} (q_k^j)^{\#,m-1})))$. Note that ψ_{m+1} is obtained from ψ_m by replacing $q^{\#,m-1}$ by $q^{\#,m}$. Since $\mathcal{S}_\vee$, $\mathcal{S}_\wedge$ and $\mathcal{P}$ are monotonic in $\sqsubseteq$ (from requirement 2.2a), and $q^{\#,m-1} \sqsubseteq q^{\#,m}$ (by induction hypothesis), $p^{\#,m+1} = \psi_{m+1} \Rightarrow \psi_m = p^{\#,m}$. ■

The termination theorem (theorem 2.2), which is restated below, can now be established.

Theorem A.5 *For every predicate p in the program, $p^{\#}$ can be effectively computed.*

Proof: Since the range of $\mathcal{P}$ has no infinite ascending chain, (requirement 2.2b), the chain $p^{\#,0}, p^{\#,1}, \ldots$ reaches a limit after a finite number of steps. Furthermore, since $\mathcal{I}$ is complete with respect to $\sqsubseteq$ (by definition) we can compute the smallest i for which $p^{\#,i+1} \sqsubseteq p^{\#,i}$, and due to lemma A.4 effectively compute the limit. ■

B Proofs for the Boolean Simplification Layer

B.1 Soundness

We first establish the soundness of equations 6, 7, 9, and 10 in figure 2 through the following lemmas. We prove that the base cases of these equations are sound and that each equation preserves soundness. Hence, in the following, an operation f is sound means that f is sound whenever it is computable. We establish the computability of these operations later in this section.

Lemma B.1 conj_quotient *and* cons_quotient *are sound and complete. That is,*
$\forall\psi\ [(\psi_2 \wedge \psi) \Rightarrow \psi_1] \Leftrightarrow [\psi \Rightarrow \text{conj\_quotient}(\psi_1, \psi_2)]$, *and*
$\forall\psi\ [(\psi_2 \wedge \psi) \Rightarrow C] \Leftrightarrow [\psi \Rightarrow \text{cons\_quotient}(C, \psi_2)]$.

Proof: To show that equation 6a preserves soundness and completeness, let $\psi' = \mathit{cons\_quotient}(C, \psi_2)$ and $\psi'' = \mathit{conj\_quotient}(\psi_1, \psi_2)$. From the soundness and completeness assumptions on the rhs, we have, $\forall\psi\ [(\psi_2 \wedge \psi \Rightarrow C) \Leftrightarrow \psi \Rightarrow \psi']$ and $\forall\psi\ [(\psi_2 \wedge \psi \Rightarrow \psi_1) \Leftrightarrow \psi \Rightarrow \psi'']$. Then, $\forall\psi\ [(\psi_2 \wedge \psi \Rightarrow \psi_1 \wedge C) \Leftrightarrow (\psi_2 \wedge \psi \Rightarrow C) \wedge (\psi_2 \wedge \psi \Rightarrow \psi_1) \Leftrightarrow (\psi \Rightarrow \psi') \wedge (\psi \Rightarrow \psi'') \Leftrightarrow (\psi \Rightarrow \psi' \wedge \psi'')]$.

That equation 6b is sound and complete is trivial to establish since $\forall\psi\ [\psi_2 \wedge \psi \Rightarrow \mathit{True} \Leftrightarrow \psi \Rightarrow \mathit{True}]$.

To show that equation 7a preserves soundness and completeness, we consider the following two cases:
Case 1: $C/C_1 = True$: From requirement 3.2 we have $C_1 \Rightarrow C$. Hence $\forall\psi' \, [C_1 \wedge \psi \wedge \psi' \Leftrightarrow C]$ and it follows that $\forall\psi' \, [(C_1 \wedge \psi \wedge \psi' \Rightarrow C) \Leftrightarrow \psi' \Rightarrow True]$.
Case 2: $C/C_1 \neq C$: Let $C/C_1 = \psi_1$ and $conj\_quotient(\psi_1, C_1 \wedge \psi) = \psi_2$. Hence, $\forall\psi' \, [(C_1 \wedge \psi \wedge \psi' \Rightarrow \psi_1) \Leftrightarrow \psi' \Rightarrow \psi_2]$. Since $\forall\psi'' \, [(C_1 \wedge \psi'' \Rightarrow C) \Leftrightarrow \psi'' \Rightarrow \psi_1]$, we have, $\forall\psi' \, [(\psi' \Rightarrow \psi_2) \Leftrightarrow (C_1 \wedge \psi \wedge \psi' \Rightarrow \psi_1) \Leftrightarrow (C_1 \wedge C_1 \wedge \psi \wedge \psi' \Rightarrow C) \Leftrightarrow (C_1 \wedge \psi \wedge \psi' \Rightarrow C)]$.

Observe that equation 7b is used only when $\forall C_1 \in \psi \; C/C_1 = C$. Hence, $\forall C_1 \in \psi \; \forall\psi' \, [(C_1 \wedge \psi' \Rightarrow C) \Leftrightarrow \psi' \Rightarrow C]$, from requirement 3.2, and it follows that $\forall\psi' \, [(\psi \wedge \psi' \Rightarrow C) \Leftrightarrow \psi' \Rightarrow C]$. ■

Lemma B.2 conj_product *and* cons_product *are sound. That is,*
$\psi_1 \wedge \psi_2 \Rightarrow \mathrm{conj\_product}(\psi_1, \psi_2)$ *and*
$\psi_1 \wedge C \Rightarrow \mathrm{conj\_product}(\psi_1, C)$.

Proof: We assume that the *conj_product* and *cons_product* appearing on the rhs of equations 9 and 10 are sound and show that every application of the equations preserve soundness.

Soundness of equations 9b and 10b are trivial since $\psi_1 \wedge True \Rightarrow \psi_1$ and $True \wedge C \Rightarrow C$.

In equation 9a the soundness assumption on the rhs means that $\psi_1 \wedge (C \wedge \psi_2) = \psi_1 \wedge \psi_2 \wedge C \Rightarrow conj\_product(\psi_1, \psi_2) \wedge C \Rightarrow cons\_product(conj\_product(\psi_1, \psi_2), C) = conj\_product(\psi_1, C \wedge \psi_2)$.

Now we show that every application of equation 10a preserves soundness. If $C/C_1 = True$ (first alternative) then clearly $(C_1 \wedge \psi) \wedge C \Rightarrow C \wedge \psi = cons\_product(C_1 \wedge \psi, C)$. If C and C_1 are orthogonal (second alternative) then $(C_1 \wedge \psi) \wedge C = C_1 \wedge (\psi \wedge C) \Rightarrow C_1 \wedge cons\_product(\psi, C) = cons\_product(C_1 \wedge \psi, C)$. If C is independent of C_1 (third alternative) then $(C_1 \wedge \psi) \wedge C = C_1 \wedge (\psi \wedge C) \Rightarrow cons\_product(\psi, C) \wedge C_1 \Rightarrow cons\_product(cons\_product(\psi, C), C_1) = cons\_product(C_1 \wedge \psi, C)$. Otherwise (fourth alternative), $(C_1 \wedge \psi) \wedge C = \psi \wedge (C \wedge C_1) \Rightarrow \psi \wedge (C * C_1)$ (from requirement 3.1) $\Rightarrow conj\_product(\psi, C * C_1) = cons\_product(C_1 \wedge \psi, C)$. ■

We can now establish the soundness of equations 5 and 8.

Lemma B.3 absorb *and* Π *are sound.*

Proof: Soundness of *absorb* (i.e., $\varphi \Rightarrow absorb(\varphi)$) is easily proved by induction on the number of conjuncts in φ and the soundness of *conj_quotient* (lemma B.1). For induction step, observe that $conj\_quotient(\psi_2, \psi_1) = True \Leftrightarrow \psi_1 \Rightarrow \psi_2$ and hence $\psi_1 \vee \psi_2 = \psi_2$.

Soundness of Π (i.e., $\varphi_1 \wedge \varphi_2 \Rightarrow \Pi(\varphi_1, \varphi_2)$) is also easily obtained by induction on the number of conjuncts in φ_1 and φ_2 and the soundness of *conj_product* (lemma B.2). ■

The Soundness Theorem (theorem 3.1), restated below, can now be established.

Theorem B.4 (Soundness) *$\mathcal{S}$, $\mathcal{P}$ and $\mathcal{I}$ are sound.*

Proof: Soundness of *absorb* and Π (lemma B.3) directly lead to soundness of $\mathcal{S}$.

Soundness of $\mathcal{P}$ follows directly from the soundness of $\mathcal{P}_\wedge$ (requirement 3.5) and the soundness of *absorb*. (lemma B.3). ■

B.2 Termination

Lemma B.5 *There is a partial order $\sqsubseteq$ such that $\mathcal{I}$ is complete w.r.t. $\sqsubseteq$.*

Proof: Define $\sqsubseteq$ to coincide with $\Rightarrow$ over all conjunctions. Note that for conjunctions, *conj_quotient* and hence $\mathcal{I}$ are complete w.r.t. $\Rightarrow$ and therefore w.r.t. $\sqsubseteq$.

Now, define $\sqsubseteq$ over disjunctions as follows: $\varphi_1 \sqsubseteq \varphi_2$ iff $\forall \psi_1 \in \varphi_1\ \exists \psi_2 \in \varphi_2\ [\psi_1 \sqsubseteq \psi_2]$. Note that $\sqsubseteq$ is Hoare's powerdomain ordering (see, *e.g.*, [11]). Transitivity and antisymmetry of $\sqsubseteq$ are straightforward. Since the definition of $\mathcal{I}$ mimics the definition of $\sqsubseteq$ and *conj_quotient* is complete w.r.t. $\sqsubseteq$, $\mathcal{I}$ is also complete w.r.t. $\sqsubseteq$. ■

Note that since $\sqsubseteq$ has been defined to be coincident with $\Rightarrow$ over conjunctions, monotonicity of operations over conjunctions can be proved using either partial order.

Lemma B.6 *Quotient (/) is monotonic in its first argument.*

Proof: Let $C_1/C = \psi_1$ and $C_2/C = \psi_2$. Hence, from requirement 3.2, we have, $\forall \psi\ [(C \wedge \psi \Rightarrow C_1) \Leftrightarrow (\psi \Rightarrow \psi_1)]$ and $\forall \psi\ [(C \wedge \psi \Rightarrow C_2) \Leftrightarrow (\psi \Rightarrow \psi_2)]$. Let $C_1 \Rightarrow C_2$. Then, $\forall \psi\ [(\psi \Rightarrow \psi_1) \Rightarrow (\psi \Rightarrow \psi_2)]$ from the definition of ψ_1 and ψ_2, and hence $\psi_1 \Rightarrow \psi_2$. ■

Lemma B.7 cons_quotient *and* conj_quotient *are monotonic in their first argument.*

Proof: Equations 6b and 7b clearly define operators that are monotonic in their first argument. Equation 6a preserves monotonicity follows since $\wedge$ is monotonic. Furthermore, equation 7a clearly preserves monotonicity since the change to the first argument uses / which, by lemma B.6, is monotonic in its first argument. ■

Lemma B.8 cons_product *and* conj_product *are monotonic.*

Proof: Equations 9b and 10b clearly define a monotonic operator. Since composition of two monotonic operators is monotonic, equation 9a and 10a define a monotonic operator whenever the operations on the rhs are monotonic. Since $*$ and $\wedge$ are monotonic (requirement 3.3), it follows that equations 9a and 10a preserve monotonicity. ■

Lemma B.9 absorb *and* Π *are monotonic with respect to* $\sqsubseteq$.

Proof: Follows from lemmas B.7 and B.8. ■

Lemma B.10 conj_quotient *and* conj_product *are computable.*

Proof: Note that *conj_quotient* and *conj_product* are defined using equations; hence computability means existence of a terminating computational procedure that implements the equations.

Note that there is a procedure that computes *cons_quotient* such that there are no direct calls to *cons_quotient*. The only non-trivial case arises when $C/C_1 \neq C$, when *conj_quotient* is invoked. Hence *cons_quotient* terminates whenever *conj_quotient* terminates.

For termination of *conj_quotient*, observe that subsequent calls to *conj_quotient* are such that

1. The number of atomic constraints in the first argument is strictly decreasing. This case arises for all direct calls to *conj_quotient*.

2. The first argument of the resultant call (via *cons_quotient*) is a conjunct ψ' such that $\forall C' \in \psi'\ [C' <_{wfo} C]$, for some C in the first argument of the original call, from requirement 3.4.

It is easy to construct an irreflexive, well founded partial order, $\prec$, based on the above observations such that every subsequent calls to *conj_quotient* strictly decreases the first argument with respect to $\prec$.

Termination of *conj_product* can be established along the same lines as that for *conj_quotient*, using the requirement 3.3. ■

Lemma B.11 absorb *and* Π *are computable.*

Proof: Follows readily from the computability of *conj_quotient* and *conj_product* (lemma B.10). ■

This leads us to the following Termination theorem (theorem 3.2) restated below:

Theorem B.12 (Termination) *There exists a partial order* $\sqsubseteq$ *such that* $\mathcal{I}$ *is complete w.r.t* $\sqsubseteq$ *and* $\mathcal{S}$ *and* $\mathcal{P}$ *are monotonic w.r.t.* $\sqsubseteq$*. Furthermore the range of* $\mathcal{P}$ *is finite.*

Proof: That $\mathcal{I}$ is complete w.r.t. $\sqsubseteq$ has been established by lemma B.5.

Monotonicity of $\mathcal{S}$ follows from the monotonicity of *absorb* and Π (lemma B.9).

$\mathcal{P}$ is monotonic due to the monotonicity of *absorb* and $\mathcal{P}_\wedge$. Furthermore, the finite ascending chain property of $\mathcal{P}$ follows from finite ascending chain property of $\mathcal{P}_\wedge$ (requirement 3.5).

Computability of all four operators is assured by the computability of *absorb*, Π (lemma B.11) and *conj_quotient* (lemma B.10). ■

C Rules for Product (Atomic Constraints Layer)

$$
\begin{array}{lrcl}
0. & C * C & \rightarrow & C \\[4pt]
1. & C(X,t) * C'(X',t') & \rightarrow & C(X,t) \wedge C(X',t'), \quad \text{whenever } X \neq X' \\[4pt]
2a. & (X \in t) * (X = Y) & \rightarrow & (Y \in t) \wedge (X = Y) \\
2b. & (X \in d) * (X = c(Y_1,\ldots,Y_n)) & \rightarrow & (X = c(Y_1,\ldots,Y_n)) \wedge \bigwedge_{i=1}^{n} (Y_i \in \mathit{subset}(d,c,i)) \\
2c. & (X \in c(\ldots)) * (X = c'(\ldots)) & \rightarrow & \mathit{False}, \quad \text{if } c \neq c' \\
2d. & (X \in c(d_1,\ldots,d_n)) * (X = c(Y_1,\ldots,Y_n)) & \rightarrow & (X = c(Y_1,\ldots,Y_n)) \wedge \bigwedge_{i=1}^{n} (Y_i \in d_i) \\[4pt]
3a. & (X = Y) * (X = t) & \rightarrow & (X = Y) \wedge (Y = t) \\
3b. & (X = t) * (Y = t) & \rightarrow & (X = Y) \wedge (Y = t) \quad \text{if } X > Y \\
 & & & (Y = X) \wedge (X = t) \quad \text{if } Y > X \\
3c. & (X = c(\ldots)) * (X = c'(\ldots)) & \rightarrow & \mathit{False}, \quad \text{if } c \neq c' \\
3d. & (X = c(Y_1,\ldots,Y_n)) * (X = c(Y_1',\ldots,Y_n')) & \rightarrow & (X = c(Y_1,\ldots,Y_n)) \wedge \bigwedge_{i=1}^{n} (Y_i = Y_i') \\[4pt]
4a. & (X \in d_1) * (X \in d_2) & \rightarrow & (X \in \mathit{glb}(d_1,d_2)) \\
4b. & (X \in c(d_1,\ldots,d_n)) * (X \in d') & \rightarrow & (X \in c(d_1'',\ldots,d_n'')), \\
 & & & \quad \text{where } d_i'' = \mathit{glb}(d_i, \mathit{subset}(d',c,i)) \ 1 \leq i \leq n \\
4c. & (X \in c(\ldots)) * (X \in c'(\ldots)) & \rightarrow & \mathit{False}, \quad \text{if } c \neq c' \\
4d. & (X \in c(d_1,\ldots,d_n)) * (X \in c(d_1',\ldots,d_n')) & \rightarrow & (X \in c(d_1'',\ldots,d_n'')), \\
 & & & \quad \text{where } d_i'' = \mathit{glb}(d_i, d_i') \ 1 \leq i \leq n
\end{array}
$$

D Formulating Sharing Constraints

We now illustrate how new constraints can be added to the atomic constraints layer, by adding constraints to capture sharing information. Sharing is represented by the atomic constraint $(X \sim S)$

where S is the set of variables that share a variable with X. Note that S is the sharing group for X, in the terminology of Jacobs and Langen [14].

We now describe the rules for computing quotient and product for sharing constraints. These rules are added to the quotient and product rules for equality and membership constraints, specified in the atomic constraints layer (figure 3 and section C of Appendix). Observe that the interaction of sharing constraints with membership and equality constraints completely defined by these rules.

The quotient operation is very simple and is described by the following rules:

$$\begin{array}{lrcl}
1a. & (X \sim S)/(X \sim S') & \rightarrow & (X \sim S - S') \\
1b. & C/(X \sim S) & \rightarrow & C \\
2a. & (Z \sim S \cup \{X\})/(X = Y) & \rightarrow & (Z \sim S \cup \{Y\}) \\
2b. & (Z \sim S)/C & \rightarrow & (Z \sim S)
\end{array}$$

The definition of *product* uses a domain dependent function *subsup*, to be supplied by the domain layer. The function $subsup(d)$ is specified using the following property: Let t be some term such that $t \in d$, and s be a subterm of t. Let t' be any term containing s. Now, if $subsup(d) = d'$ then $t' \in d'$. For example, let the domain be $\{g, f, u\}$ where g represents the set of all ground terms, f represents set of all terms with functor root (i.e., non-variable terms) and u be the universe of terms. Then, $subsup(g) = f$, $subsup(f) = f$ and $subsup(u) = u$.

The rules for *product* are:

$$\begin{array}{lrcl}
1a. & (X = Y) * (X \sim S) & \rightarrow & (X = Y) \wedge (Y \sim S - \{Y\}) \\
1b. & (X = Y) * (Y \sim S) & \rightarrow & (X = Y) \wedge (Y \sim S)) \\
1c. & (X = Y) * (Z \sim S \cup \{X\}) & \rightarrow & (X = Y) \wedge (Z \sim S \cup \{X, Y\}) \\
1d. & (X = Y) * (Z \sim S) & \rightarrow & (X = Y) \wedge (Z \sim S) \\
2a. & (X = t) * (X \sim S) & \rightarrow & (X = t) \wedge (X \sim S) \\
2b. & (X = c(Y_1, \ldots, Y_n)) * (Y_i \sim S) & \rightarrow & (X = c(Y_1, \ldots, Y_n)) \wedge (X \sim S) \wedge \bigwedge_{i=1}^{n}(Y_i \sim S) \\
2c. & (X = c(Y_1, \ldots, Y_n)) * (Z \sim S \cup \{Y_i\}) & \rightarrow & (X = c(Y_1, \ldots, Y_n)) \wedge (Z \sim S \cup \{X, Y_i\}) \\
2d. & (X = t) * (Z \sim S) & \rightarrow & (X = t) \wedge (Z \sim S) \\
3a. & (X \sim \{Y_1, \ldots, Y_n\}) * (X \in d) & \rightarrow & (X \sim \{Y_1, \ldots, Y_n\}) \wedge (X \in d) \wedge \bigwedge_{i=1}^{n}(Y_i \in subsup(d)) \\
3b. & (X \sim \{Y_1, \ldots, Y_n\}) * (Y_i \in d) & \rightarrow & (X \sim \{Y_1, \ldots, Y_n\}) \wedge (X \in subsup(d)) \wedge \bigwedge_{i=1}^{n}(Y_i \in d) \\
4a. & (X \sim S) * (X \sim S') & \rightarrow & (X \sim S \cup S') \\
4b. & (X \sim S) * (Y \sim S') & \rightarrow & (X \sim S) \wedge (Y \sim S')
\end{array}$$

E An Example of Type Analysis – *append*

We provide a detailed illustration of the type analysis, formulated in section 5, on the *append* predicate (from page 10). The abstract equation for *append* is

$$\begin{array}{rl}
append^{\#}(X, Y, Z) & = \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}(\quad (X = [\,] \wedge Y = Z) \quad \vee \\
& \qquad\qquad (X = [U|V] \wedge Z = [U|W] \wedge append^{\#}(V, Y, W))))
\end{array}$$

Since *append* is recursive, we have $append^{\#,0} = False$. The fix-point computation proceeds in the following sequence.

$$\begin{array}{rcl}
append^{\#,0} & = & False \\
append^{\#,1} & = & \mathcal{P}^{\{X,Y,Z\}}(X = [\,] \wedge Y = Z) \\
& = & (X = [\,] \wedge Y = Z) \\
append^{\#,2} & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_{\vee}((X = [\,] \wedge Y = Z), \\
& & \quad \mathcal{S}_{\wedge}((X = [U|V] \wedge Z = [U|W]), (V = [\,] \wedge Y = W))))
\end{array}$$

$$
\begin{array}{rcl}
 & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_\vee((X = [\,] \wedge Y = Z), (X = [U|V] \wedge V = [\,] \wedge Z = [U|Y]))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}((X = [\,] \wedge Y = Z) \vee (X = [U|V] \wedge V = [\,] \wedge Z = [U|Y])) \\
 & = & \mathit{absorb}((X = [\,] \wedge Y = Z) \vee (X \in \mathit{list})) \\
 & = & (X \in \mathit{list}) \\
\mathit{append}^{\#,3} & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_\vee((X = [\,] \wedge Y = Z), \\
 & & \quad \mathcal{S}_\wedge((X = [U|V] \wedge Z = [U|W]), ((V = [\,] \wedge Y = W) \vee (V \in \mathit{list}))))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_\vee((X = [\,] \wedge Y = Z), \\
 & & \quad \mathit{absorb}(\prod((X = [U|V] \wedge Z = [U|W]), ((V = [\,] \wedge Y = W) \vee (V \in \mathit{list})))))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_\vee((X = [\,] \wedge Y = Z), \\
 & & \quad \mathit{absorb}((X = [U|V] \wedge V = [\,] \wedge Z = [U|Y]) \vee (X = [U|V] \wedge V \in \mathit{list} \wedge Z = [U|W])))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}(\mathcal{S}_\vee((X = [\,] \wedge Y = Z), (X = [U|V] \wedge V \in \mathit{list} \wedge Z = [U|W]))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}(\mathit{absorb}((X = [\,] \wedge Y = Z) \vee (X = [U|V] \wedge V \in \mathit{list} \wedge Z = [U|W]))) \\
 & = & \mathcal{P}^{\{X,Y,Z\}}((X = [\,] \wedge Y = Z) \vee (X = [U|V] \wedge V \in \mathit{list} \wedge Z = [U|W])) \\
 & = & \mathit{absorb}((X = [\,] \wedge Y = Z) \vee (X \in \mathit{list})) \\
 & = & (X \in \mathit{list})
\end{array}
$$