Constraint Systems for Pattern Analysis of Constraint Logic-Based Languages

Roberto Bagnara
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
bagnara@di.unipi.it
Phone: 050/887267 Fax: 050/887226

Abstract

Pattern analysis consists in determining the shape of the set of solutions of the constraint store at some program points. Our basic claim is that pattern analyses can all be described within a unified framework of constraint domains. We show the basic blocks of such a framework as well as construction techniques which induce a hierarchy of domains. In particular, we propose a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the product operation in a completely homogeneous way. That is achieved by regarding semantic domains as particular kinds of (ask-and-tell) constraint systems. These constraint systems allow to express communication among domains in a very simple way. The techniques we propose allow for smooth integration within an appropriate framework for the definition of non-standard semantics of constraint logic-based languages. The effectiveness of this methodology is being demonstrated by a prototype implementation of CHINA, a CLP($\mathcal{H}$, N) analyzer we have developed.

Keywords: Constraint Systems, Constraint-based Languages, Data-flow Analysis, Abstract Interpretation.

1 Introduction

Pattern analysis for constraint logic-based languages consists in determining the shape of the set of solutions of the constraint store at some program point. For usual applications (most prominently, program specialization) the interesting program points are procedure calls and procedure (successful) exits.

In the case of Prolog, pattern analysis has been extensively studied (see [9] for a summary of this work). In the case of CLP, besides the generalization to CLP($\mathcal{H}$) of the ideas and techniques used for Prolog, not much has been done. A key observation here is that the shape of solutions can be conveniently described by constraints. Thus the CLP framework is general enough to encompass (some of) its own data-flow analyses. Intuitively, this is done by replacing the standard constraint domain with one suitable for expressing the desired information. This fundamental aspect was brought to light in [5] and elaborated in [12].

For languages of the kind of CLP(N), where N is some numerical domains, the first steps towards pattern analysis were moved in [3, 4]. [2] describes some of the more important applications of such analyses. The work done in this field is being generalized to CLP($\mathcal{H}$, N) languages, integrating numerical and symbolic pattern analysis. This is done with a variety of techniques, including depth-k abstraction. A more restricted kind of integration has recently been described in [17]. Here, the numerical part is essentially the one proposed in [3].

Now, instead of directly describing the techniques employed in [3, 4, 2, 17], we concentrate on what is missing from them: a general notion of constraint domain which allows one to adequately describe both the "logical part" of concrete computations (e.g. answer constraints) and as much pattern analyses (e.g. the shape of those answer constraints) we can think about.

We believe that it is possible to describe every pattern analysis within a unified framework of constraint domains. In particular we wish the framework being able to accommodate approximate inference techniques whose importance relies on very practical considerations, such as representing good compromises between precision and computational efficiency. Some of these techniques will be sketched in the sequel.

Then, what will be needed is a generalized algebraic semantics for constraint logic programs, parameterized with respect to an underlying constraint domain. The main advantages of this approach [12] are that: (1) different instances of CLP can be used to define non-standard semantics for constraint logic programs; and (2) the abstract interpretation of CLP programs can be thus formalized inside the CLP paradigm.

Let us concentrate on constraint domains for pattern analysis. They are algebraic structures of the kind

$$\bar{\mathcal{D}} = \left\langle \mathcal{D}, \preceq, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}, \{d_{\bar{X}\bar{Y}}\} \right\rangle, \tag{1}$$

where[1] $\mathcal{D}$ is the set of constraints expressing the properties of interest. $\mathcal{D}$ is partially ordered with respect to $\preceq$ which, intuitively, relates the information content of constraints: $C_1 \preceq C_2$ means that "C_1 is more precise than C_2". $\otimes$ and $\oplus$ are binary operators modeling conjunction and (weak) disjunction. $\{\exists_\Delta\}$ is a family of unary operators, indexed over finite subsets of variables, modeling projection of constraints onto designated sets of variables. $\mathbf{0}$ and $\mathbf{1}$ represent, intuitively, the class of unsatisfiable constraints and the class of non-constraints (i.e. those which do not provide any information), respectively. The family of distinguished elements $\{d_{\bar{X}\bar{Y}}\}$, indexed on pairs of n-tuple of variables, allows to model parameter passing.

In this setting, data-flow analysis is then performed (or at least justified) through abstract interpretation [8, 9], i.e., "mimicking" the program run-time behavior by "executing" it, in a finite way, on an approximated (abstract) constraint domain. We will thus have two constraint domains of the form (1): the "concrete" and the "abstract" one. Following a generalized semantic approach, the concrete and abstract semantics are more easily related, being instances (over two different constraint systems) of the same generalized semantics, which is entirely parametric on a constraint domain. Thus, to ensure correctness, it will be sufficient to exhibit an "abstraction function" α which is a semimorphism between the constraint domains [10].

In this paper we describe a hierarchy of constraint systems which capture all the pattern analyses we know of, as well as the "concrete" collecting semantics they abstract. The basis is constituted by a set of finite constraints, each expressing some partial information about a program execution's state. Once this is given (simple constraint systems, Section 2), we provide standard ways of representing and composing finite constraints (determinate constraint systems, Section 3). Then we can have the notion of *dependency* built into the constraint system (ask-and-tell constraint systems Section 4). Another construction is the one which allows us to treat *disjunction* (powerset constraint systems, Section 5). Finally, in Section 6 we sketch how to achieve combination of domains by considering dependencies within *product constraint systems*. We feel that, indeed, this is one of more important contributions of this paper.

For the sake of simplicity we will present constraint systems omitting the distinguished elements modeling parameter passing. For most applications $d_{\bar{X}\bar{Y}}$ is simply a constraint expressing some sort of equivalence between X and Y. We disregard them also because, differently from [12], we do not require them to satisfy any interesting algebraic property.

[1] For space reasons we omit many details.

2 Simple constraint systems

The basic blocks of our construction are *simple constraint systems* (or s.c.s.), very similar to those of [19], but with a *totally uninformative* token ($\top$) as in [20].

Definition 2.1 (Simple constraint system.) *A* simple constraint system *is a structure* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$, *where* $\mathcal{C}$ *is a set of (not better specified) constraints,* $\bot \in \mathcal{C}$, $\top \in \mathcal{C}$, *and* $\vdash \subseteq \wp_f(\mathcal{C}) \times \mathcal{C}$ *is a compact* entailment relation *such that, for each* $C, C' \in \wp_f(\mathcal{C})$ *and* $c, c' \in \mathcal{C}$*:*

$$\begin{array}{llll} E_1. & c \in C \;\Rightarrow\; C \vdash c, & E_3. & (C \vdash c) \wedge (\forall c' \in C : C' \vdash c') \;\Rightarrow\; C' \vdash c, \\ E_2. & C \vdash \top, & E_4. & \{\bot\} \vdash c \end{array}$$

We consider also the extension $\vdash \subseteq \wp(\mathcal{C}) \times \wp(\mathcal{C})$ *such that, for each* $C, C' \in \wp(\mathcal{C})$,

$$C \vdash C' \quad \Leftrightarrow \quad \forall c' \in C' : \exists C'' \subseteq_f C \,.\, C'' \vdash c'.$$

It is clear that condition E_1 implies reflexivity of $\vdash$, while condition E_3 amounts to transitivity. E_2 qualifies $\top$ as the least informative token: it will be needed just as a "marker" when the *product* of simple constraint systems will be considered (see Section 6). E_4 ensures that $\mathcal{C}$ is a finitely generable element.

In general, describing the "standard" semantics of a CLP(X) language is an easy matter. Let T be the theory which corresponds to the domain X [15]. Let D be an appropriate set of formulas in the vocabulary of T closed under conjunction and existential quantification. Define $\Gamma \vdash c$ iff Γ entails c in the logic, with non-logical axioms T. Then $(D, \vdash)$ is the required simple constraint system. For CLP($\mathcal{H}$) (i.e. pure Prolog) one takes the Clark's theory of equality. For CLP($\mathbb{R}$) the theory RCF of real closed fields will do the job.

However, describing "standard" constraint domains is not the reason which motivated our work. Here are the original motivations.

2.1 Pattern analysis for numeric domains

The analysis described in [3, 4, 2] is based on constraint inference (a variant of constraint propagation) [11]. This technique, developed in the field of artificial intelligence, has been applied to temporal and spatial reasoning [1, 21].

Let us focus our attention to arithmetic domains, where the constraints are binary relations over expressions. Let E be the set of arithmetic expressions of interest and I the set of intervals over some computable set of numbers (e.g. rational or floating point numbers). Then our constraints are given by

$$\mathcal{C} = \big\{\, e_1 \bowtie e_2 \;\big|\; \bowtie \in \{=, \neq, \leq <, \geq, >\}, e_1, e_2 \in E \,\big\} \cup \{\, e \lhd I \mid e \in E, I \in \mathsf{I} \,\}.$$

The meaning of the constraint $e \lhd I$ is the obvious one: any value the expression e can take is contained in I. Thus $\mathcal{C}$ provides a mixture of qualitative (relationships) and quantitative (bounds) knowledge.

Now, the approximate inference techniques we are interested in can be encoded into a consequence relation $\vdash$ over $\mathcal{C}$. Let us see some of them. The most trivial one is *symmetric closure*: $\{e_1 \bowtie e_2\} \vdash e_2 \bowtie^{-1} e_1$, where $\bowtie^{-1}$ is the inverse of $\bowtie$ (e.g., $<$ is the inverse of $>$, $\geq$ of $\leq$ and so on). A more interesting qualitative technique is *transitive closure*, allowing inferences like $A < C$ from $A \leq B$ and $B < C$. It is formalized by axioms of the form $\{e_1 \leq e_2, e_2 < e_3\} \vdash e_1 < e_3$. A classical quantitative technique is *interval arithmetic* which allows to infer the variation interval of an expression from the intervals of its sub-expressions. Let $f(e_1, \ldots, e_k)$ be any arithmetic expression having e_1, ..., e_k as subexpressions. Then $\big\{f(e_1, \ldots, e_k) \lhd I, e_1 \lhd I_1, \ldots, e_k \lhd I_k\big\} \vdash f(e_1, \ldots, e_k) \lhd \ddot{f}(I_1, \ldots, I_k)$, where $\ddot{f}\colon \mathsf{I}^k \to \mathsf{I}$ is such that for each $x_1 \in I_1$, ..., $x_k \in I_k$, $f(x_1, \ldots, x_k) \in \ddot{f}(I_1, \ldots, I_k)$. An example inference is: $A \lhd [3, 6) \wedge B \lhd [-1, 5] \vdash A + B \lhd [2, 11)$.

Another technique is *numeric constraint propagation*, which consists in determining the relationship between two expressions when their associated intervals do not overlap, except possibly at their endpoints. The associated family of axioms is $\{e_1 \lhd I_1, e_2 \lhd I_2\} \vdash e_1 \bowtie e_2$, with the side condition $\forall x_1 \in I_1, x_2 \in I_2 : x_1 \bowtie x_2$. For example, if $A \in (-\infty, 2]$, $B \in [2, +\infty)$, and $C \in [5, 10]$, we can infer that $A \leq B$ and $A < C$. It is also possible to go the other way around, i.e., knowing that $U < V$ may allow to refine the intervals associated to U and V so that they do not overlap. We call this *weak interval refinement*: $\{e_1 \bowtie e_2, e_1 \lhd I_1, e_2 \lhd I_2\} \vdash e_1 \lhd I_1'$, where I_1' is obtained by shrinking I_1 so to ensure that $x_1 \in I_1'$ iff $x_1 \in I_1 \wedge \exists x_2 \in I_2 \,.\, x_1 \bowtie x_2$.

In summary, by considering the transitive closure of $\vdash$ and with some minor technical additions we end up with a simple constraint system which characterizes precisely the combination of the above (and possibly other) techniques.

3 Determinate constraint systems

By axioms E_1 and E_3 of Definition 2.1 the entailment relation of a simple constraint system is a preorder. Now, instead of considering the quotient poset with respect to the induced equivalence relation, a particular choice of the equivalence classes' representatives is made: closed sets with respect to entailment. This representation is a very convenient domain-independent strong normal form for constraints.

Definition 3.1 (Elements.) [19] *The* elements *of an s.c.s.* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *are the* entailment-closed *subsets of* $\mathcal{C}$*, that is, those* $C \subseteq \mathcal{C}$ *such that* $\exists C' \subseteq_f C \,.\, C' \vdash c$ *implies* $c \in C$*. The set of elements of* $\langle \mathcal{C}, \vdash \rangle$ *is denoted by* $|\mathcal{C}|$*.*

The poset of elements is thus given by $\langle |\mathcal{C}|, \supseteq \rangle$. Notice that we deviate from [19] in that we order our constraint systems in the dual way.

Definition 3.2 (Inference map, finite elements.) *Given a simple constraint system* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$*, the* inference map *of* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *is the function* $\rho \colon \wp(\mathcal{C}) \to \wp(\mathcal{C})$ *given, for each* $C \subseteq \mathcal{C}$*, by* $\rho(C) = \{\, c \mid \exists C' \subseteq_f C \,.\, C' \vdash c \,\}$*. It is well known that* ρ *is a kernel operator, over the complete lattice* $\langle \wp(\mathcal{C}), \supseteq \rangle$*, whose image is* $|\mathcal{C}|$*. The image of the restriction of* ρ *onto* $\wp_f(\mathcal{C})$ *is denoted by* $|\mathcal{C}|_0$*. Elements of* $|\mathcal{C}|_0$ *are called* finitely generated constraints *or simply* finite constraints*.*

From here on we will only work with finitely generated constraints, since we are not concerned with infinite behavior of CLP programs. The next step in our construction is about *determinate constraint systems* (or d.c.s.).

Definition 3.3 (Determinate constraint system.) *Let* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$ *be a simple constraint system. Let* $\mathbf{0}, \mathbf{1} \in |\mathcal{C}|_0$*,* $\otimes \colon |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$*, and* $\vdash \subseteq |\mathcal{C}|_0 \times |\mathcal{C}|_0$ *be given, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, by*

$$\begin{aligned} \mathbf{0} &= \mathcal{C}, & \qquad C_1 \otimes C_2 &= \rho(C_1 \cup C_2), \\ \mathbf{1} &= \rho(\emptyset), & \qquad C_1 \vdash C_2 &\Leftrightarrow C_1 \otimes C_2 = C_1. \end{aligned}$$

The projection operators $\exists_\Delta \colon |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *are given, for each* $\Delta \subseteq_f Vars$ *and each* $C \in \wp(\mathcal{C})$*, by*

$$\exists_\Delta C = \rho\bigl(\{\, c \in C \bigm| FV(c) \subseteq \Delta \,\}\bigr).$$

Finally, let $\oplus \colon |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *be an operator enjoying the following properties:*

J_1. $\langle |\mathcal{C}|_0, \oplus, \mathbf{0} \rangle$ *is a commutative and idempotent monoid;*

J_2. *for each* $C_1, C_2 \in |\mathcal{C}|_0$*,* $C_1 \vdash C_1 \oplus C_2$ *and* $C_2 \vdash C_1 \oplus C_2$*.*

We will refer to the structure $\langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}\rangle$ *as the* determinate constraint system *over* $\mathcal{S}$ *and* $\oplus$*. The relation* $\sqsubseteq$ *induced by* $\oplus$ *over* $|\mathcal{C}|_0$ *is given, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, by* $C_1 \sqsubseteq C_2$ *iff* $C_1 \oplus C_2 = C_2$*. The relations* $\vdash$ *and* $\sqsubseteq$ *are referred to, respectively, as the* approximation ordering *and the* computational ordering *of the determinate constraint system.*

Observe that the required conditions on $\oplus$ are quite reasonable. The purpose of $\oplus$ is that of "merging" the information originating from different paths in the semantic construction. In this view, axiom J_1 is very natural: associativity and commutativity amount to say that we can merge paths in any order, idempotence means that we do not loose precision blindly, and $\mathbf{0}$ being the monoid unit accounts for the ability of discarding failed computation paths. Condition J_2 states the correctness of the merge operation, characterizing it as a (not necessarily least) upper bound operator with respect to the approximation ordering.

Notice that the distinction between approximation ordering and computational ordering is important. We assume that our semantics are defined as (approximations of) least fixpoints of some operator[2] ϕ. So, while the approximation ordering, in general abstract interpretation, specifies the relative precision of program properties (e.g. entailment of constraints in our particular case), the computational ordering holds among the iterates $\phi^k(\perp)$ during the fixpoint computation. The case where the two orderings coincide (e.g. in [12]) is thus to be considered a special one. In our treatment, keeping them distinct allows for more freedom in the choice of the merge operator.

Since the set of finite computation paths is, in general, denumerably infinite, we consider also the following strengthening of Definition 3.3.

Definition 3.4 (Closed d.c.s.) *A d.c.s.* $\langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}\rangle$ *is said* closed *iff it satisfies*

- J_3. *for each family* $\{C_i \in |\mathcal{C}|_0\}_{i\in\mathbb{N}}$, $\bigoplus_{i\in\mathbb{N}} C_i = C_1 \oplus C_2 \oplus \cdots$ *exists and is unique in* $|\mathcal{C}|_0$*; moreover, associativity, commutativity, and idempotence of* $\oplus$ *apply to denumerable as well as to finite families of operands.*

So, the operation of merging together the information coming from all the computation paths always makes sense in a closed determinate constraint system. Notice however that property J_3 is only necessary when the semantic construction requires it. This will never happen when considering "abstract" semantic constructions formalizing data-flow analyses (which are finite in nature). In these cases the idea of merging infinitely many pieces of information is a nonsense in itself.

Determinate constraint systems enjoy several properties. Here are some elementary ones: $\sqsubseteq$ is a partial order and $C_1 \sqsubseteq C_2$ implies $C_1 \vdash C_2$; $\otimes$ and $\oplus$ are componentwise monotone with respect to $\vdash$ and $\sqsubseteq$, respectively; $\mathbf{0}$ is an annihilator for $\otimes$, while $\mathbf{1}$ is a unit for $\otimes$ and an annihilator for $\oplus$. Finally, for absorption laws we have $C_2 = (C_1 \oplus C_2) \otimes C_2$ and $C_2 \vdash (C_1 \otimes C_2) \oplus C_2$. At a higher level, here is the situation.

Theorem 3.1 *Let* $\mathcal{D} = \langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}\rangle$ *be a determinate constraint system. Then the structure* $\langle |\mathcal{C}|_0; \vdash, \mathbf{0}, \mathbf{1}, \otimes\rangle$ *is a bounded meet-semilattice and* $\langle |\mathcal{C}|_0; \sqsubseteq, \mathbf{0}, \mathbf{1}, \oplus\rangle$ *is a join-semilattice. Moreover, if* $\mathcal{D}$ *is complete, then* $\langle |\mathcal{C}|_0; \sqsubseteq, \mathbf{0}, \mathbf{1}, \oplus\rangle$ *is a (join-) complete lattice.*

Notice that $\langle |\mathcal{C}|_0, \otimes, \oplus, \mathbf{0}, \mathbf{1}\rangle$, in general, is not a lattice. Both $\otimes$ and $\oplus$ are associative, commutative, and idempotent. But, as stated above, while one of the absorption laws holds, only one direction of the dual law is generally valid. In particular $\otimes$ is not required to be componentwise monotone with respect to $\sqsubseteq$, and $\oplus$ might be not componentwise monotone with respect to $\vdash$. Observe also that $\oplus$ does not distribute, in general, over $\otimes$, as this would imply the equivalence of the two absorption laws.

[2]For example, if we choose a bottom-up (backward) semantic construction for CLP, this will be an immediate consequence operator T_P parameterized on the underlying constraint system [12]. We disregard this issues here, as we concentrate on the construction of constraint domains.

4 Ask-and-tell constraint systems

We now consider constraint systems having additional structure. This additional structure allows to express, at the constraint system level, that the imposition of certain constraints must be delayed until some other constraints are imposed. In [18] similar constructions are called *ask-and-tell constraint systems*. In our construction, ask-and-tell constraint systems are built from determinate constraint systems by regarding some kernel operators as constraints. We follow [18] in considering cc as *the* language framework for expressing and computing with kernel operators. For this reason we will present kernel operators as cc agents. For our current purposes we only need a very simple fragment of the determinate cc language: the one of *finite* cc *agents*. This fragment is described in [19] by means of a declarative semantics. Here we give an operational characterization which is better suited to our needs.

Definition 4.1 (Finite cc agents: syntax.) *A* finite cc agent *over a simple constraint system $\mathcal{S} = \langle \mathcal{C}, \vdash, \perp, \top \rangle$ is any string generated by the following grammar:*

$$\textbf{Agent} ::= \mathrm{tell}(C) \mid \mathrm{ask}(C) \to \textbf{Agent} \mid \textbf{Agent} \parallel \textbf{Agent}$$

where $C \in |\mathcal{C}|_0$. We will denote by $\mathcal{A}(\mathcal{S})$ the language of such strings. The following explicit definition is also given:

$$\mathrm{ask}(C_1; \ldots; C_n) \to \textbf{Agent} \quad \equiv \quad \bigl(\mathrm{ask}(C_1) \to \textbf{Agent}\bigr) \parallel \cdots \parallel \bigl(\mathrm{ask}(C_n) \to \textbf{Agent}\bigr).$$

When this will not cause confusion we will freely drop the syntactic sugar, writing C and $C_1 \to C_2$ where $\mathrm{tell}(C)$ and $\mathrm{ask}(C_1) \to \mathrm{tell}(C_2)$ are intended.

The introduction of a syntactic normal form for finite cc agents allows to simplify to subsequent semantic treatment.

Definition 4.2 (Finite cc agents: syntactic normal form.) *The transformation η over $\mathcal{A}(\mathcal{S})$ is defined, for each $C^a, C_1^a, C_2^a, C^t \in |\mathcal{C}|_0$ and $A, A_1, A_2 \in \mathcal{A}(\mathcal{S})$, as follows:*

$$\begin{aligned}
\eta(C^a \to C^t) &= \begin{cases} \mathbf{1} \to \mathbf{1} & \text{if } C^a \vdash C^t, \\ C^a \to (C^a \otimes C^t) & \text{otherwise,} \end{cases} \\
\eta(C^t) &= \mathbf{1} \to C^t, \\
\eta\bigl(C_1^a \to (C_2^a \to A)\bigr) &= \eta\bigl((C_1^a \otimes C_2^a) \to A\bigr), \\
\eta\bigl(C^a \to (A_1 \parallel A_2)\bigr) &= \eta\bigl((C^a \to A_1) \parallel (C^a \to A_2)\bigr), \\
\eta(A_1 \parallel A_2) &= \eta(A_1) \parallel \eta(A_2).
\end{aligned}$$

The following fact is easily proved.

Proposition 4.1 *The transformation η of* Definition 4.2 *is well defined. Furthermore, if $A \in \mathcal{A}(\mathcal{S})$ then $\eta(A)$ is of the form $(C_1^a \to C_1^t) \parallel \cdots \parallel (C_n^a \to C_n^t)$.*

Thus, by considering only agents of the form $\|_{i=1}^n C_i^a \to C_i^t$ we do not loose any generality. We will call elementary agents of the kind $C^a \to C^t$ *ask–tell pairs*.

Now we express the operational semantics of finite cc agents by means of rewrite rules. An agent in normal form is rewritten by applying the logical rules of the calculus modulo a structural congruence. This congruence states, intuitively, that we can regard an agent as a set of (concurrent) ask–tell pairs.

Definition 4.3 (A calculus of finite cc agents.) *Let $\mathbf{1}_{\mathrm{A}} = \mathbf{1} \to \mathbf{1}$. The* structural congruence *of the calculus is the smallest congruence relation $\equiv_s$ such that $\langle \mathcal{A}(\mathcal{S}), \|, \mathbf{1}_{\mathrm{A}} \rangle_{/\equiv_s}$ is a commutative and idempotent monoid. The reduction rules of the calculus are given in* Figure 1. *We also define the relation $\rho_{\mathrm{A}} \subseteq \mathcal{A}(\mathcal{S}) \times \mathcal{A}(\mathcal{S})$ given, for each $A, A' \in \mathcal{A}(\mathcal{S})$, by*

$$A \; \rho_{\mathrm{A}} \; A' \quad \Leftrightarrow \quad \exists n \in \mathbb{N} \,.\, A = A_1 \wedge A_n = A' \wedge A_1 \mapsto A_2 \mapsto \cdots \mapsto A_n \not\mapsto$$

Structure	$\dfrac{A_1 \equiv_s A_1' \quad A_1' \longmapsto A_2' \quad A_2' \equiv_s A_2}{A_1 \longmapsto A_2}$ $\qquad \dfrac{A_1 \longmapsto A_1'}{A_1 \parallel A_2 \longmapsto A_1' \parallel A_2}$
Reduction	$\dfrac{C_2^a \vdash C_1^a \qquad C_1^t \vdash C_2^t}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto (C_1^a \to C_1^t)}$
Deduction	$\dfrac{C_1^t \vdash C_2^a}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto (C_1^a \to (C_1^t \otimes C_2^t)) \parallel (C_2^a \to C_2^t)}$
Absorption	$\dfrac{C_1^a \vdash C_2^a}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto ((C_1^a \otimes C_2^a) \to (C_1^t \otimes C_2^t)) \parallel (C_2^a \to C_2^t)}$

Figure 1: Reduction rules for finite cc agents.

In the following we will systematically abuse the notation denoting $\mathcal{A}(\mathcal{S})_{/\equiv_s}$ simply by $\mathcal{A}(\mathcal{S})$. Consequently, every assertion concerning $\mathcal{A}(\mathcal{S})$ is to be intended *modulo structural congruence.*

Proposition 4.2 *The term-rewriting system depicted in Figure 1 is strongly normalizing. Thus the relation $\rho_{\rm A}$ is indeed a function $\rho_{\rm A}: \mathcal{A}(\mathcal{S}) \to \mathcal{A}(\mathcal{S})$.*

The situation here is almost identical to the one of Definition 3.2, in that we have a domain-independent strong normal form also for the present class of constraints (i.e. agents) incorporating the notion of dependency.

Definition 4.4 (Elements.) *The* elements *of $\mathcal{A}(\mathcal{S})$ are those which are closed under (are the fixed points of) the* inference map *$\rho_{\rm A}$. The set of elements of $\mathcal{A}(\mathcal{S})$ will be denoted by $|\mathcal{A}(\mathcal{S})|$.*

The strict analogy with determinate constraint systems continues with the following.

Definition 4.5 (Ask-and-tell constraint system.) *Given a simple constraint system $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$, let $\mathcal{A} = |\mathcal{A}(\mathcal{S})|$. Then let $\mathbf{0}_{\rm A}, \mathbf{1}_{\rm A} \in \mathcal{A}$, $\otimes_{\rm A}: \mathcal{A} \times \mathcal{A} \to \mathcal{A}$, and $\vdash_{\rm A} \subseteq \mathcal{A} \times \mathcal{A}$ be given, for each $A_1, A_2 \in \mathcal{A}$, by*

$$\begin{aligned} \mathbf{0}_{\rm A} &= \mathbf{1} \to \mathbf{0}, & A_1 \otimes_{\rm A} A_2 &= \rho_{\rm A}(A_1 \parallel A_2), \\ \mathbf{1}_{\rm A} &= \mathbf{1} \to \mathbf{1}, & A_1 \vdash_{\rm A} A_2 &\Leftrightarrow A_1 \otimes_{\rm A} A_2 = A_1. \end{aligned}$$

The projection operators The projection operators $\exists_\Delta^{\rm A}: \mathcal{A} \to \mathcal{A}$ are given, for each $\Delta \subseteq_f$ Vars and $A \in \mathcal{A}$, by

$$\exists_\Delta^{\rm A} A = \rho_{\rm A}(A_{|\Delta}),$$

where

$$A_{|\Delta} = \left\{ (\exists_\Delta C^a \to \exists_\Delta C^t) \;\middle|\; \begin{array}{l} (C^t \to C^a) \in A \quad \text{and} \\ ((\mathbf{1} \to \exists_\Delta C^a) \otimes_{\rm A} A) \vdash_{\rm A} (\mathbf{1} \to C^a) \end{array} \right\}.$$

Finally, let $\oplus_{\rm A}: \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ be an operator satisfying, for each $A_1, A_2 \in \mathcal{A}$, the following axioms:

J_1^a. *$\langle \mathcal{A}, \oplus_{\rm A}, \mathbf{0}_{\rm A} \rangle$ is a commutative and idempotent monoid;*

J_2^a. *for each $A_1, A_2 \in \mathcal{A}$, $A_1 \vdash_{\rm A} A_1 \oplus_{\rm A} A_2$ and $A_2 \vdash_{\rm A} A_1 \oplus_{\rm A} A_2$.*

Again, we will denote by $\sqsubseteq_{\rm A}$ the relation induced by $\oplus_{\rm A}$ over $\mathcal{A}$: $A_1 \sqsubseteq_{\rm A} A_2$ iff $A_1 \oplus_{\rm A} A_2 = A_2$. We will refer to $\langle \mathcal{A}, \vdash_{\rm A}, \otimes_{\rm A}, \oplus_{\rm A}, \{\exists_\Delta^{\rm A}\}, \mathbf{0}_{\rm A}, \mathbf{1}_{\rm A} \rangle$ as the ask-and-tell constraint system *over $\mathcal{S}$ and $\oplus_{\rm A}$. We will call it* closed *iff it satisfies*

J_3^a. *for each family* $\{A_i \in \mathcal{A}\}_{i\in\mathbb{N}}$, $\bigoplus_{i\in\mathbb{N}}^a A_i = A_1 \oplus_{\mathrm{A}} A_2 \oplus_{\mathrm{A}} \cdots$ *exists and is unique in* $\mathcal{A}$*; moreover, associativity, commutativity, and idempotence of* $\oplus_{\mathrm{A}}$ *apply to denumerable as well as to finite families of operands.*

Once you have a determinate constraint system, you also have an ask-and-tell constraint system, whose merge operator is induced as follows.

Definition 4.6 *Let* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$ *be an s.c.s., and let* $\mathcal{D} = \big\langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1} \big\rangle$ *a d.c.s. over* $\mathcal{S}$*. Let also* $\mathcal{A} = |\mathcal{A}(\mathcal{S})|$*, and let* $\hat{\oplus}_{\mathrm{A}}: \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ *be given by*

$$\left(\mathop{\|}_{i=1}^{n} A_i\right) \hat{\oplus}_{\mathrm{A}} \left(\mathop{\|}_{j=1}^{m} B_j\right) = \rho_{\mathrm{A}}\left(\mathop{\|}_{i=1}^{n} \mathop{\|}_{j=1}^{m} (A_i \hat{\oplus}_{\mathrm{A}} B_j)\right),$$

where, for any two ask-tell pairs $C_1^a \to C_1^t$ *and* $C_2^a \to C_2^t$*, we define*

$$(C_1^a \to C_1^t) \,\hat{\oplus}_{\mathrm{A}}\, (C_2^a \to C_2^t) = \begin{cases} \mathbf{1}_{\mathrm{A}} & \text{if } C^a \vdash C^t, \\ C^a \to (C^a \otimes C^t) & \text{otherwise,} \end{cases}$$

being $C^a = C_1^a \otimes C_2^a$ *and* $C^t = C_1^t \oplus C_2^t$*. We will refer to* $\hat{\oplus}_{\mathrm{A}}$ *as* the merge operator over $\mathcal{A}$ induced by $\mathcal{D}$.

Proposition 4.3 $\big\langle \mathcal{A}, \vdash_{\mathrm{A}}, \otimes_{\mathrm{A}}, \hat{\oplus}_{\mathrm{A}}, \{\exists_\Delta^{\mathrm{A}}\}, \mathbf{0}_{\mathrm{A}}, \mathbf{1}_{\mathrm{A}} \big\rangle$ *is an ask-and-tell constraint system. Furthermore, it is closed iff* $\mathcal{D}$ *is so.*

Notice that ask-and-tell constraint systems subsume the determinate ones, where only "tells" were considered. In fact we have $\eta(C_1) \otimes_{\mathrm{A}} \eta(C_2) = \eta(C_1 \otimes C_2)$ and $\eta(C_1) \hat{\oplus}_{\mathrm{A}} \eta(C_2) = \eta(C_1 \oplus C_2)$.

It is time to start showing why we are interested in this kind of constraint systems, even though for the more exciting things we have to wait until the next section, where combination of constraint domains are introduced. Ask-and-tell constraint system are needed to model approximate inference techniques which can be very useful for pattern analysis.

4.1 More pattern analysis for numeric domains

Following Section 2.1, there is another technique which is used for the analysis described in [3, 4, 2]: *relational arithmetic* [21]. This technique allows to infer constraints on the qualitative relationship of an expression to its arguments. If we take the ask-and-tell constraint system over the simple one of Section 2.1, we can describe it by a number of (concurrent) agents. Here are some of them (where $\bowtie$ ranges in $\{=, \neq, \leq <, \geq, >\}$):

$$\begin{aligned} \mathrm{ask}(x \bowtie 0) &\;\to\; \mathrm{tell}\big((x+y) \bowtie y\big) \\ \mathrm{ask}(x > 0 \wedge y > 0 \wedge x \bowtie 1) &\;\to\; \mathrm{tell}\big((x * y) \bowtie y\big) \\ \mathrm{ask}(x \bowtie y) &\;\to\; \mathrm{tell}(e^x \bowtie e^y) \end{aligned}$$

An example of inference is deducing $X + 1 \leq Y + 2X + 1$ from $X \geq 0 \wedge Y \geq 0$. Notice that there is no restriction to linear constraints.

5 Powerset constraint systems

For the purpose of pattern analysis it is not necessary to represent the "real disjunction" of constraints collected through different computation paths, since we are interested in the common information only. To this end, a weaker notion of disjunction suffices. We define *powerset constraint systems*, which are instances of a well known construction, i.e., disjunctive completion[3] [10]. When this is applied to a simple constraint system $\mathcal{S}$ it yields the following.

[3]Given a poset $\langle L, \bot, \leq \rangle$, the relation $\preceq \subseteq \wp(L) \times \wp(L)$ induced by $\leq$ is given, for each $M_1, M_2 \in \wp(L)$ by $(M_1 \preceq M_2) \Leftrightarrow (\forall m_1 \in M_1 : \exists m_2 \in M_2 \,.\, m_1 \leq m_2)$. A subset $M \in \wp(L)$ is said *non-redundant* iff $\bot \notin M$ and $\forall m_1, m_2 \in M : m_1 \leq m_2 \Rightarrow m_1 = m_2$ The set of non-redundant subsets of L is denoted by $\wp_n(L)$. The function $\Omega: \wp(L) \to \wp_n(L)$ is given, for each $M \in \wp(L)$, by $\Omega(M) = M \setminus \{\, m \in M \mid m = \bot \vee \exists m' \in M \,.\, m < m' \,\}$.

Definition 5.1 (Powerset constraint system.) *Given an s.c.s.* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$, *the* powerset constraint system *over* $\langle \mathcal{C}, \vdash \rangle$ *is given by* $\big\langle \wp_n\big(|\mathcal{C}|_0\big), \vdash_{\mathrm{P}}, \otimes_{\mathrm{P}}, \oplus_{\mathrm{P}}, \{\exists^{\mathrm{P}}_{\Delta}\}, \mathbf{0}_{\mathrm{P}}, \mathbf{1}_{\mathrm{P}} \big\rangle$, *where*

$$\begin{aligned} \mathbf{0}_{\mathrm{P}} &= \emptyset, & \exists^{\mathrm{P}}_{\Delta} S &= \Omega\big(\{\, \exists_{\Delta} C \mid C \in S \,\}\big), \\ \mathbf{1}_{\mathrm{P}} &= \{\mathbf{1}\}, & S_1 \vdash_{\mathrm{P}} S_2 &\Leftrightarrow \forall C_1 \in S_1 : \exists C_2 \in S_2 \,.\, C_1 \vdash C_2, \\ S_1 \oplus_{\mathrm{P}} S_2 &= \Omega(S_1 \cup S_2), & S_1 \otimes_{\mathrm{P}} S_2 &= \Omega\big(\{\, C_1 \otimes C_2 \mid C_1 \in S_1, C_2 \in S_2 \,\}\big). \end{aligned}$$

With these definitions $\big\langle \wp_n\big(|\mathcal{C}|_0\big); \vdash_{\mathrm{P}}, \mathbf{0}_{\mathrm{P}}, \mathbf{1}_{\mathrm{P}}, \otimes_{\mathrm{P}}, \oplus_{\mathrm{P}} \big\rangle$, is a join-complete, distributive bounded lattice. We can of course apply the powerset construction also to ask-and-tell constraint systems.

6 Combination of domains

It is well known that different data-flow analyses can be combined together. In the framework of abstract interpretation this can be achieved by means of standard constructions such as reduced product and down-set completion [8, 9]. The key point is that the combined analysis can be more precise than each of the component ones for they can mutually improve each other. However, the degree of cross-fertilization is highly dependent on the degree and quality of interaction taking place among the component domains.

We now propose a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the "meet operation" in a completely homogeneous way.

This is achieved by considering ask-and-tell constraint systems built over *product* simple constraint systems. These constraint systems allow to express communication among domains in a very simple way. They also inherit all the semantic elegance of concurrent constraint programming languages, which provide the basis for their construction. Recently, a methodology for the combination of abstract domains has been proposed in [7], which is directly based onto low level actions such as *tests* and *queries*. While the approach in [7] is immediately applicable to an apparently wider range of analyses (this is one subject for further study), the approach we follow here for pattern analysis has the merit of being much more elegant.

We start with a set of simple constraint systems $\big\{\, \langle \mathcal{C}_i, \vdash_i, \bot_i, \top_i \rangle \;\big|\; i = 1, \ldots, n \,\big\}$, each expressing some properties of interest, and we wish to combine them so to: (1) perform all the analyses at the same time; and (2) have the domains cooperate to the intent of mutually improving each other. The first goal is achieved by considering the product of the simple constraint systems.

Definition 6.1 (Product of simple constraint systems.) *Given a finite family of simple constraint systems* $\mathcal{S}_i = \langle \mathcal{C}_i, \vdash_i, \bot_i, \top_i \rangle$ *for* $i = 1, \ldots, n$, *the* product *of the family is the structure given by* $\prod_{i=1}^{n} \mathcal{S}_i = \langle \mathcal{C}_\times, \vdash_\times, \bot_\times, \top_\times \rangle$, *where the product tokens are*

$$\mathcal{C}_\times = \big\{\, (c_1, \top_2, \ldots, \top_n) \;\big|\; c_1 \in \mathcal{C}_1 \,\big\} \cup \cdots \cup \big\{\, (\top_1, \ldots, \top_{n-1}, c_n) \;\big|\; c_n \in \mathcal{C}_n \,\big\} \cup \{\bot_\times\},$$

the product entailment is defined, for each $C \in \wp_f(\mathcal{C}_\times)$, *by*

$$\begin{array}{lllcll} C & \vdash_\times & (c_1, \top_2, \ldots, \top_n) & \Leftrightarrow & \Pi_1(C) & \vdash_1 & c_1, \\ & \vdots & & \vdots & & \vdots & \\ C & \vdash_\times & (\top_1, \ldots, \top_{n-1}, c_n) & \Leftrightarrow & \Pi_n(C) & \vdash_n & c_n, \end{array}$$

where, for each $i = 1, \ldots, n$, $\Pi_i \colon \wp(\mathcal{C}_\times) \to \mathcal{C}_i$ *is the obvious projection mapping a set of* n*-tuples onto the set of* i*-th components. Finally,* $\bot_\times = (\bot_1, \ldots, \bot_n)$ *and* $\top_\times = (\top_1, \ldots, \top_n)$.

If you had a family of determinate constraint systems $\mathcal{D}_i$ built on top of the $\mathcal{S}_i$'s, you can easily "recycle" the merge operators $\oplus_i$ to obtain a merge operator $\oplus_\times \colon |\mathcal{C}_\times|_0 \times |\mathcal{C}_\times|_0 \to |\mathcal{C}_\times|_0$ which allows you to build a product d.c.s.

So, taking the product of constraint systems, we have realized the simplest form of domain combination. It corresponds to the direct product construction of [8], allowing for different analyses to be carried out at the same time. Notice that there is no communication at all among the domains.

However, as soon as we consider the ask-and-tell constraint system built over the product, we can express asynchronous communication among the domains in complete freedom. At the very least we would like to have the *smash product* among the component domains. This is realized by the agent $\|_{i=1}^{n} \mathbf{0}_i \to \mathbf{0}_\times$. To say it operationally, the *smash* agent globalizes the (local) failure on any component domain. This is the only domain-independent agent we have.

Things become much more interesting when instantiated over particular constraint domains. In the CLP($\mathcal{R}$) system [16] non-linear constraints (e.g. $X = Y * Z$) are delayed (i.e. not treated by the constraint solver) until they become linear (e.g. until either Y or Z are constrained to take a single value). In standard semantic treatments this is modeled in the operational semantics by carrying over, besides the sequence of goals yet to be solved, a set of delayed constraints. Constraints are taken out from this set (and incorporated into the constraint store) as soon as they become linear.

We believe that this can be viewed in an alternative way which is more elegant, as it easily allows for taking into account the delay mechanism also in the fixpoint semantics, and makes sense from an implementation point of view. The basic claim is the following: CLP($\mathcal{R}$) has *three* computation domains: Herbrand, $\mathbb{R}$ (well, an approximation of it), and *definiteness*.

In other words, it also manipulates, besides the usual ones, constraints of the kind $X = \mathit{gnd}^\flat$ which is interpreted as the variable X being definitively bound to a unique value. We can express the semantics of CLP($\mathcal{R}$) (at a certain level of abstraction) with delay of non-linear constraints by considering the ask-and-tell constraint system over the product of the above three domains. In this view, a constraint of the form $X = Y * Z$ in a program actually corresponds to the agent

$$\mathrm{ask}(Y = \mathit{gnd}^\flat; Z = \mathit{gnd}^\flat) \to \mathrm{tell}(X = Y * Z).$$

In fact, any CLP($\mathcal{R}$) user *must* know that $X = Y * Z$ is just a shorthand for that agent! A similar treatment could be done for logic programs with delay declarations.

Obviously, this cannot be forgotten in abstract constraint systems intended to formalize correct data-flow analyses of CLP($\mathcal{R}$). Referring back to sections 2.1 and 4.1, when the abstract constraint system extracts information from non-linear constraints, i.e. $\mathrm{ask}(Y > 0 \wedge Z > 0 \wedge Y \bowtie 1) \to \mathrm{tell}\big((Y * Z) \bowtie Z\big)$ by relational arithmetic, you cannot simply let $X = Y * Z$ stand by itself. By doing this you would incur the risk of *overshooting* the concrete constraint system (thus loosing soundness), which is unable to deduce anything from non-linear constraints. The right thing to do is to combine your abstract constraint system with one for definiteness (by the product and the ask-and-tell construction) and considering, for example, the following agent:

$$\begin{aligned} \mathrm{ask}(Y = \mathit{gnd}^\sharp; Z = \mathit{gnd}^\sharp) \quad &\to \quad \mathrm{ask}(Y > 0 \wedge Z > 0 \wedge Y \bowtie 1) \\ &\to \quad \mathrm{tell}\big((Y * Z) \bowtie Z\big) \end{aligned}$$

Beware not to confuse $X = \mathit{gnd}^\flat$ with $X = \mathit{gnd}^\sharp$. The first is the *concrete one*: X is definite if and only if $X = \mathit{gnd}^\flat$ is entailed in the current store. In contrast, having $X = \mathit{gnd}^\sharp$ entailed in the current *abstract* constraint store means that X is certainly bound to a unique value in the concrete computation, but this is only a sufficient condition, not a necessary one.

Let us see another example. The analysis described in [13] aims at the compile-time detection of those non-linear constraints that will become linear at run time. This analysis is important for remedying the limitation of CLP($\mathcal{R}$) to linear constraints by incorporating powerful (and computationally complex) methods from computer algebra as the ones employed in RISC-CLP(Real) [14]. With the results of the above analysis this extension can be done in a smooth way: non-linear constraints which are guaranteed to become linear will be simply delayed, while only the other non-linear constraints will be treated with the special solving techniques. Thus, programs not requiring the extra power of these techniques will be hopefully recognized as such, and will not pay any penalties. The analysis of [13] is a kind of definiteness. One of its difficulties shows up when considering the simplest non-linear constraint: $X = Y * Z$. Clearly X is definite if Y and Z are such. But we cannot conclude that the definiteness of Y follows from the one of X and Z, as we need also the condition $Z \neq 0$. Similarly, we would like to conclude that X is definite if Y or Z have a zero value. Thus we need approximations of the concrete values of variables (i.e. pattern analysis), something which is not captured by common definiteness analyses while being crucial when dealing with non-linear constraints. Then, just take the combination to obtain something like[4]

$$\begin{array}{ll} & \text{ask}(Y = gnd^\sharp \wedge Z = gnd^\sharp) \rightarrow \text{tell}(X = gnd^\sharp) \\ \| & \text{ask}(Y = 0; Z = 0) \rightarrow \text{tell}(X = gnd^\sharp) \\ \| & \text{ask}(X = gnd^\sharp \wedge Z = gnd^\sharp \wedge Z \neq 0) \rightarrow \text{tell}(Y = gnd^\sharp) \\ \| & \text{ask}(X = gnd^\sharp \wedge Y = gnd^\sharp \wedge Y \neq 0) \rightarrow \text{tell}(Z = gnd^\sharp) \end{array}$$

7 Conclusion and future work

We have shown a hierarchy of constraint systems which, both theoretically and experimentally, have several nice features. One feature we did not mention before is that proving two members of the hierarchy being one a correct approximation of the other is often quite easy.

Almost all of the ideas in this paper have been satisfactorily implemented in the CHINA analyzer [2]. The experimental results obtained with the implementation represent a strong encouragement to proceed along these lines.

In particular, we have proposed a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the domains' operations in a completely homogeneous way. This is achieved by regarding semantic domains as particular kinds of (ask-and-tell) constraint systems. These constraint systems allow to express communication among domains in a very simple way. They also inherit all the semantic elegance of concurrent constraint programming languages, which provide the basis for their construction. Future work includes answering the following questions: are there variation of these ideas which are applicable also to analysis oriented towards "non-logical" properties? That is, properties which are not preserved as the computation progresses? Can we turn this constructions capturing dependence, combination, and disjunction into an algebra of constraint domains?

References

[1] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *CACM*, 26(11):832–843, 1983.

[2] R. Bagnara. On the detection of implicit and redundant numeric constraints in CLP programs. In *Proc. GULP–PRODE'94*, 1994.

[3] R. Bagnara, R. Giacobazzi, and G. Levi. Static Analysis of CLP Programs over Numeric Domains. In M. Billaud *et al.*, editors, *Actes WSA'92*, volume 81–82 of *Bigre*, pages 43–50, 1992.

[4] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-Flow Analysis. In *Proc. IEEE CAIA'93*, pages 270–276, 1993. IEEE Press.

[5] P. Codognet and G. Filè. Computations, Abstractions and Constraints. In *Proc. Fourth IEEE Int'l Conference on Computer Languages*. IEEE Press, 1992.

[4] Notice that this is much more precise than the Prop formula $X \leftarrow Y \wedge Z$ [6].

[6] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. LICS'91*, pages 322–327. IEEE Press, 1991.

[7] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Proc. POPL'94*, pages 227–239, 1994.

[8] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL'79*, pages 269–282, 1979.

[9] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[10] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992.

[11] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32:281–331, 1987.

[12] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. FGCS'92*, pages 581–591, 1992.

[13] M. Hanus. Analysis of nonlinear constraints in CLP($\mathcal{R}$). In D. S. Warren, editor, *Proc. ICLP'93*, pages 83–99. The MIT Press, 1993.

[14] H. Hong. RISC-CLP(Real): Logic Programming with Non-linear Constraints over the Reals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. The MIT Press, 1993.

[15] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. POPL'87*, pages 111–119. ACM, 1987.

[16] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM TOPLAS'92*, 14(3):339–395, 1992.

[17] G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H, N). In M. Hermenegildo and J. Penjam, editors, *Proc. PLILP'94*, volume 844 of *LNCS*, pages 400–414. Springer-Verlag, 1994.

[18] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[19] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. POPL'91*, pages 333–353. ACM, 1991.

[20] D. Scott. Domains for Denotational Semantics. In M. Nielsen and E. M. Schmidt, editors, *Proc. ICALP'82*, volume 140 of *LNCS*, pages 577–613. Springer-Verlag, 1982.

[21] R. Simmons. Commonsense Arithmetic Reasoning. In *Proc. AAAI-86*, pages 118–124, 1986.