A hierarchy of Constraint Systems for Data-Flow Analysis of Constraint Logic-Based Languages

Roberto Bagnara

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy

Many interesting analyses for constraint logic-based languages are aimed at the detection of *monotonic* properties, that is to say, properties which are preserved as the computation progresses. Our basic claim is that most, if not all, of these analyses can be described within a unified notion of constraint domains. We present a class of constraint systems which allows for a smooth integration within an appropriate framework for the definition of non-standard semantics of constraint logic-based languages. Such a framework is also presented and motivated. We then show how such domains can be built, as well as construction techniques which induce a hierarchy of domains with interesting properties. In particular, we propose a general methodology for domain combination with asynchronous interaction (i.e. the interaction is not necessarily synchronized with the domains' operations). Following this methodology, interesting combinations of domains can be expressed with all the the semantic elegance of concurrent constraint programming languages.

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

1 Introduction

Many interesting and useful data-flow analyses for constraint logic-based languages are aimed at the detection of *monotonic* properties, that is to say, properties which are preserved as the computation progresses. They usually consist in determining the *shape* of the set of solutions of the constraint store at some program points. Analyses which fall in this category include definiteness (or groundness), symbolic patterns, types, numerical bounds and relations, symbolic size-relations and so on. The typical examples of non-monotonic properties are freeness and aliasing. A key observation is that monotonic properties can be conveniently expressed by constraints, which are then accumulated in the analysis process much in the same way as it happens during the "concrete"

executions. Thus, frameworks of constraint-based languages are, in principle, general enough to encompass several of their 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, for the case of CLP, in [9] and elaborated in [21]. In [21] it is presented a generalized algebraic semantics for constraint logic programs, which is parameterized with respect to an underlying constraint domain. The main advantages of this approach are that: (1) different instances of CLP can be used to define non-standard semantics for constraint logic programs; and (2) several abstract interpretations of CLP programs can be thus formalized inside the CLP paradigm. In this setting, data-flow analysis is then performed (or at least justified) through abstract interpretation [13,14], i.e. "mimicking" the program run-time behavior by "executing" it, in a finite way, on an approximated (abstract) constraint domain. 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 semi-morphism between the constraint domains [7,15].

We move our steps from [21] by providing a more general notion of constraint domain which allows one to adequately describe both the "logical part" of concrete computations (i.e. answer constraints) and all the monotonic abstract interpretations we know of. In particular our notion of constraint system is 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 examples. The new notion of constraint domain requires the introduction of a new generalized semantics framework which is more liberal than the one of [21].

Moreover, and here comes the main point, we show that our constraint domains admit interesting constructions. The most important one consists in upgrading a domain so that it will be able to represent and manipulate *dependencies* among constraints. This is done by regarding a restricted class of cc agents as constraints. This construction, among other things, opens up the possibility of combining domains in a novel and interesting way. Following this methodology, the asynchronous interaction between domains can be expressed with all the elegance which derives from the cc framework.

For space reasons we have omitted many details. The interested reader is referred to [4], a much longer version of this paper including all the proofs and more examples. The plan of the paper is as follows: Section 2 introduces some basic notions and notations used throughout the paper. Section 3 explains our generalized semantics for CLP languages, as well as the abstract interpretation

framework we employ. Section 4 introduces *simple constraint systems*: some important building blocks of the hierarchy. Section 5 builds on the previous one presenting standard ways of representing and composing finite constraints: *determinate constraint systems*. In Section 6 it is shown how a constraint system is upgraded to incorporate a weak form of disjunction (suitable to monotonic properties) by means of *powerset constraint systems*. Section 7 presents a different kind of upgrade: the one needed in order to have the notion of *dependency* built into the constraint system. This is done considering *ask-and-tell constraint systems*. Section 8 deals with the interesting problem of combining domains. A technique is shown which consists in applying the *ask-and-tell construction* to *product constraint systems*. We feel that, indeed, this is one of the more important contributions of this work. Finally, Section 9 draws some conclusions and presents some directions for further study.

2 Preliminaries

Throughout the paper we will assume familiarity with the basic notions of lattice theory, semantics of logic programming languages, and abstract interpretation.

Let U be a set. The set of all subsets of U will be denoted by $\wp(U)$. The set of all *finite* subsets of U will be denoted by $\wp_f(U)$. The notation $S \subseteq_f T$ stands for $S \in \wp_f(T)$. For $S \subseteq U$ we will denote the complement $U \setminus S$ by $\overline{S}$, when U is clear from the context. For $S, T \subseteq U$ the notation $S \uplus T$ denotes *disjoint union*, emphasizing the fact that $S \cap T = \varnothing$. By $U^\star$ we will denote the set of all finite sequences of elements drawn from U. The empty sequence is denoted by ε. For $x \in U^\star$, the *length* of x will be denoted by $\# x$, and, for i such that $1 \leq i \leq \# x$, the notation $x[i]$ stands for the i-th element of x. Let $S_1, \ldots, S_n$ be sets. We will denote elements of $S_1 \times \cdots \times S_n$ by $\langle e_1, \ldots, e_n \rangle$. The *projection mappings* $\pi_i \colon S_1 \times \cdots \times S_n \to S_i$ are defined, for $i = 1, \ldots, n$, by $\pi_i(\langle e_1, \ldots, e_n \rangle) \stackrel{\text{def}}{=} e_i$. The *liftings* $\pi_i \colon \wp(S_1 \times \cdots \times S_n) \to \wp(S_i)$ given by $\pi_i(T) \stackrel{\text{def}}{=} \{\, \pi_i(t) \mid t \in T \,\}$ will also be used.

A *partial order* $\preceq$ over a set P is a binary relation which is reflexive, transitive, and antisymmetric. $\preceq$ is a *total order* if, in addition, for each $x, y \in P$, either $x \preceq y$ or $y \preceq x$. A set P equipped with a partial (resp. total) order $\preceq$ is said to be *partially ordered* (resp. *totally ordered*), and sometimes written $\langle P, \preceq \rangle$. Partially ordered sets are also called *posets*. A subset S of a poset $\langle P, \preceq \rangle$ is said to be a *chain* if it is totally ordered wrt $\preceq$. Given $x \in P$, the *downward closure* of x in $\langle P, \preceq \rangle$ is given by $\{\, y \in P \mid y \preceq x \,\}$ and denoted by $\downarrow x$. Given a poset $\langle P, \preceq \rangle$ and $S \subseteq P$, $y \in P$ is an *upper bound* for S iff $x \preceq y$ for each $x \in S$. An upper bound y for S is the *least upper bound* (or *lub*) of S iff for every upper bound y' for S it is $y \preceq y'$. The *lub*, when it exists, is unique.

In this case we write $y = lub\ S$. *Lower bounds* and *greatest lower bounds* are defined dually. $\langle P, \preceq \rangle$ is said *bounded* if it has a minimum and a maximum element. A monotone and idempotent self-map $\rho\colon P \to P$ over a poset $\langle P, \preceq \rangle$ is a *kernel operator* (or *lower closure operator*) if it is *reductive*, that is to say $\forall x \in P : \rho(x) \preceq x$.

A poset $\langle L, \preceq \rangle$ such that, for each $x, y \in L$, both $lub\{x, y\}$ and $glb\{x, y\}$ exist, is called a *lattice*. In this case, *lub* and *glb* are also called, respectively, the *join* and the *meet* operations of the lattice. A poset where only the *glb* operation is well-defined is called a *meet-semilattice*. A *complete lattice* is a lattice $\langle L, \preceq \rangle$ such that every subset of L has both a least upper bound and a greatest lower bound.

An algebra $\langle L, \wedge, \vee \rangle$ is also called a *lattice* if $\wedge$ and $\vee$ are two binary operations over L which are commutative, associative, idempotent, and satisfy the following *absorption laws*, for each $x, y \in L$: $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$. The two definitions of lattices are equivalent. This can be seen by setting up the isomorphism given by: $x \preceq y \overset{\text{def}}{\iff} x \wedge y = x \overset{\text{def}}{\iff} x \vee y = y$, $glb\{x, y\} \overset{\text{def}}{=} x \wedge y$, and $lub\{x, y\} \overset{\text{def}}{=} x \vee y$.

The notions of meet-semilattice and of bounded lattice are imported into the algebraic definition in the natural way. For instance, a *bounded lattice* is an algebra $\langle L, \wedge, \vee, \bot, \top \rangle$ such that $\langle L, \wedge, \vee \rangle$ is a lattice and the following two *annihilation* laws are satisfied for each $x \in L$: $x \wedge \bot = \bot$ and $x \vee \top = \top$. A lattice is said *distributive* if it satisfies, for each $x, y, z \in L$, the *distributive laws* $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$. It is well known that, for lattices, the distributive laws are equivalent.

An algebra $\langle M, \cdot, 1 \rangle$ is a *monoid* iff '$\cdot$' is an associative binary operator over M, and $1 \in M$ satisfies the *identity law*: for each $x \in M$, $x \cdot 1 = 1 \cdot x = x$. The monoid $\langle M, \cdot, 1 \rangle$ is said *commutative* or *idempotent* if '$\cdot$' is so.

The sets of all natural, real, and ordinal numbers will be denoted, respectively, by $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{O}$. The first limit ordinal equipotent with the set of natural numbers is denoted by ω.

3 A case study: CLP

The constraint domains which are the subject of this work are not bound to a particular class of constraint logic-based languages. However, for the sake of clarity and to help the intuition, we will focus on the class of CLP languages [26,28], which is more and more influential and captures several existing, implemented languages. We will present a *generalized approach* to the semantics

of CLP programs of which abstract interpretation is an important instance. This is necessary for a full understanding of how the domains of later sections are employed in data-flow analysis of CLP languages.

3.1 CLP: the syntax

Here we give a precise definition of what we will call a CLP(C) program. Notice that here we are concerned with syntax only. In general, C (the language of *atomic constraints*) is a subset of a first order language L (the language of constraints). Let us start by defining L itself.

Definition 1 (Language of constraints.) *Let V and Λ be two disjoint denumerable sets of variable symbols. Let us also fix two particular isomorphisms between Λ and $\mathbb{N}$, and between V and $\mathbb{N}$. Let $Vars \stackrel{\text{def}}{=} V \uplus \Lambda$. Let Ω and Π_C be two finite sets of operation and predicate symbols, respectively, each symbol being characterized with its* arity. *Let also $Vars$, Ω and Π_C be mutually disjoint. $\mathsf{L} = \mathsf{L}(Vars, \Omega, \Pi_C, \ldots)$ is a* language of constraints *iff it is any first order language with equality built (by means of standard constructions, possibly with connectives and quantifiers) over the given sets of symbols.*

(The extra-set of variables Λ allows us to simplify the following treatment. We will stipulate that the *heads* of clauses can only contain variable's symbols drawn from Λ.) Now, a CLP language can impose restrictions on the form of constraints that may actually appear in programs. However these restrictions must not destroy too much of the language's expressivity.

Definition 2 (Atomic constraint.) *Given a language of constraints L, any subset C of L is a* language of atomic constraints *if*

(i) it is closed under variable renaming; and
(ii) it contains all the formulas of the form $X = t$, where $X \in Vars$ and t is a term built over Ω and $Vars$.

Before introducing the full syntax of CLP programs a few remarks about notation are in order. We will denote program variables by means of capital letters. Tuples of distinct variable will be denoted by $\bar{X}$, $\bar{Y}$, and so forth. Tuples are always assumed to be of the right cardinality, e.g., if p is a predicate symbol of arity n and we write $p(\bar{X})$, then $\bar{X}$ is an n-tuple. Special tuples denoted by $\vec{\Lambda}$, denoting initial finite segments of Λ, will also be used (notice that we have fixed a total ordering on Λ). In the above hypotheses, by writing $p(\vec{\Lambda})$ we understand that $\vec{\Lambda}$ denotes the n-tuple consisting of the first n variable's symbols in Λ. We will also abuse the notation by confusing a tuple with the set of its components. We can now introduce the notion of CLP(C) program.

Definition 3 (CLP program.) *Let* C *be a language of atomic constraints with distinguished variable symbols in* Λ. *Let* Π_P *be a finite set of predicate symbols, disjoint from the symbols used in* C. *We will denote by* A_P *the set of atoms over* Π_P, *that is*

$$\mathsf{A}_P \stackrel{\text{def}}{=} \left\{ q(\bar{X}) \;\middle|\; q \in \Pi_P, \bar{X} \in \mathit{Vars}^\star \right\}.$$

A CLP(C) *program* P *over* Π_P *is a finite sequence of* clauses *of the form*

$$p(\vec{\Lambda}) :- \langle b_1, \ldots, b_k \rangle, \quad \textit{with } p \in \Pi_P \textit{ and } k \geq 0,$$

where, for $1 \leq i \leq k$, $b_i \in \mathsf{C} \cup \mathsf{A}_P$ *and* $\mathit{vars}(b_i) \cap \Lambda \subseteq \vec{\Lambda}$. $p(\vec{\Lambda})$ *is called the* head *of the rule, whereas* $\langle b_1, \ldots, b_k \rangle$ *is the* body.

The syntax of any CLP language can be defined in such a way, by augmenting the first order language on which it is based with the set of distinguished variable symbols Λ and transforming each program along the lines of Definition 3 by means of standard techniques. This transformation is always possible by virtue of Definition 2. This *normalization* of programs has the property that predicate's symbols are always applied to tuples of distinct variables. Further, all the heads of the rules defining a program predicate p/n have the same variables in the same positions. Observe that the body B of a clause $p(\vec{\Lambda}) :- B$ is a sequence, that is an element of $(\mathsf{C} \cup \mathsf{A}_P)^\star$. As programs themselves are sequences, the semantic constructions will be free to take into account the selection and search rules used in real languages. So far for the syntax, we now examine the (possibly non-standard) semantics of CLP languages.

3.2 Non-standard semantics for CLP

Here we start from very basic facts. We recognize the existence of four different activities in the execution, and thus in the analysis, of constraint logic programs:

(i) different execution paths are explored;
(ii) along any path, constraints are accumulated in the so called *constraint store*;
(iii) the constraint store is recursively subdivided into parts. The activity of imposing restrictions in the way different parts can interact is usually called *hiding*.
(iv) Pieces of information (parameters) are passed between program rules.

In a generalized semantics setting [21] these activities are captured by algebraic structures of the form

$$\left\langle \mathcal{D}, \trianglelefteq, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{\Lambda}}\}_{\vec{\Lambda} \in \Lambda^\star}, \{d_{\bar{X}\bar{Y}}\}_{\bar{X},\bar{Y} \in Vars^\star} \right\rangle,$$

The $\oplus$ operator models the merging of information coming from different execution paths. The $\mathbf{0}$ elements represent the information content of an empty (or failed) execution path. The $\otimes$ operator models the constraint accumulation process. The element $\mathbf{1}$ stands, intuitively, for the empty constraint store, i.e. the one containing no information at all. The $\bar{\exists}_{\vec{\Lambda}}$ operators represent the *hiding* process: any variable $X \notin \vec{\Lambda}$ appearing in the scope of $\bar{\exists}_{\vec{\Lambda}}$ is isolated (hidden) from other occurrences of X outside the scope. The "complement sign" which appears on top of $\bar{\exists}_{\vec{\Lambda}}$ signifies that we formalize hiding in a dual way with respect to traditional approaches [35,21]. The so-called *diagonal elements* $d_{\bar{X}\bar{Y}}$ represent, roughly speaking, the fact that the tuples of variables $\bar{X}$ and $\bar{Y}$ are tightly correlated with respect to the properties of interest. The relation $\trianglelefteq$ specifies the relative precision of program properties. $D_1 \trianglelefteq D_2$ means that "D_1 is more precise than D_2". In other words, every set of computations which enjoys property D_1 also enjoys property D_2. In the framework of abstract interpretation, $\trianglelefteq$ is referred to as the *approximation ordering* of the domain [15].

The way we define domains of interpretation for CLP languages is clearly highly dependent on the application we have in mind. For our current purposes we restrict ourselves to a specific class of domains which is obtained by imposing restrictions on the general scheme (see [4]). For a full understanding of the hypotheses which are behind our approach, we now spell out the above mentioned restrictions. This makes clear what is the class of properties which are captured by the hierarchy of domains that will be presented later.

- We are interested properties of programs which are valid for all *terminating* and *successful* computations;
- we are neither interested in the *order* in which computations are taken, nor in their *multiplicities*;
- we focus on *monotonic* properties, that is, those which are preserved as computation progresses;
- further, we restrict out interest to *logical* properties. This means, roughly speaking, that $\otimes$ is interpreted as logical conjunction.

Now the question is: how do we represent the properties of interest? A simple, but far reaching answer was first given in [9]: we can represent properties by means of constraints. This opens up the possibility of computing non-standard semantics of CLP, and, in particular, abstract interpretations, *within* the CLP framework. In this setting the result of the abstract interpretation of a CLP program P is obtained by "executing" (in a finite way) another CLP program P', strongly related to P, over a non-standard domain. Intuitively, this is done by

replacing the standard constraint domain with one suitable for expressing the desired information. This possibility led to the idea of a generalized semantics for CLP programs, proposed in [21]. A generalized semantics is parameterized over the (possibly non-standard) constraint system which constitute the domain of the computation. The main advantages of this approach are that:

(i) different instances of CLP can be used to define non-standard semantics for constraint logic programs;
(ii) the semantics of these instances are all captured within a unified algebraic framework; and, in particular,
(iii) many relevant abstract interpretations of CLP programs can be formalized inside the CLP paradigm.

The next section is devoted to the class of domains outlined above.

3.3 Constraint systems

Since we aim at a pervasive treatment, we would like to avoid talking too much about what a constraint is. However, we cannot overlook some basic facts on the relationship between constraints and program variables. The purpose of constraints is, roughly speaking, to restrict the range of values variables can take. For our present objectives the following definition suffices.

Definition 4 (Constraint.) *The class of* constraints *is defined by:*

(i) a well-formed formula of any first-order language $\mathcal{L}$ with variables' symbols in Vars is a constraint;
(ii) any set of constraints is a constraint;
(iii) any (meta-level) predicate p/n applied to n constraints is a constraint;
(iv) nothing is a constraint if not by virtue of (i), (ii), and (iii).

Now, on the relationship between constraints and variables, we can reason inductively as follows.

Definition 5 (Variables, free variables, and renaming.) *With reference to* Definition 4*, if c is a constraint by virtue of (i) then the notions of* variables of c *and of* free variables of c *are assumed as primitive. These sets of variables are denoted, respectively, by $vars(c)$ and $FV(c)$. An invertible and idempotent mapping from and to variable's symbols which is the identity almost everywhere is called* renaming*. We will use the notation $[\bar{Y}/\bar{X}]$ for renamings, where $\bar{Y}$ and $\bar{X}$ are disjoint tuples of distinct variables. The renaming $[Y/X]$ has no effect on c if $X \notin FV(C)$, whereas variables' capture is avoided by consistent renaming of bound variables. Besides that, the constraint $c[\bar{Y}/\bar{X}]$ is assumed as defined.*

If C is a constraint because of (ii) then $vars(C) \stackrel{\text{def}}{=} \bigcup_{c \in C} vars(c)$ *and*[1]

$$FV(C) \stackrel{\text{def}}{=} \Big\{ X \in Vars \;\Big|\; \exists c \in C \,.\, \exists Y \in Vars \,.\, c[Y/X] \notin C \Big\}. \tag{1}$$

With these definitions the notion of renaming for C is extended as expected. The application of a renaming to C is defined element-wise.

If $\mu(C_1, \ldots, C_n)$ is a constraint by virtue of (iii) then the above notions are extended as they would be in any first-order language.

In the sequel we will apply renamings carefully so that, when we write $C[\bar{Y}/\bar{X}]$, it is ensured that $FV(C) \cap \bar{Y} = \varnothing$. We will emphasize this fact by saying that $[\bar{Y}/\bar{X}]$ is a renaming *for* C.

All the members of our hierarchy of domains will turn out to be *constraint systems* in the precise sense stated by the following definition.

Definition 6 (Constraint system.) *Any algebra $\bar{\mathcal{D}}$ of the form*

$$\Big\langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{\Lambda}}\}_{\vec{\Lambda} \in \Lambda^\star}, \{d_{\bar{X}\bar{Y}}\}_{\bar{X},\bar{Y} \in Vars^\star} \Big\rangle$$

is a constraint system *if and only if it satisfies the following conditions:*

G_0. *$\mathcal{D}$ is a set of constraints;*
G_1. *$\langle \mathcal{D}, \otimes, \mathbf{1} \rangle$ is a commutative and idempotent monoid;*
G_2. *$\langle \mathcal{D}, \oplus, \mathbf{0} \rangle$ is a commutative and idempotent monoid;*
G_3. *$\mathbf{0}$ is an* annihilator *for $\otimes$, i.e., for each $C \in \mathcal{D}$, $C \otimes \mathbf{0} = \mathbf{0}$;*
G_4. *for each $C_1, C_2 \in \mathcal{D}$, $C_1 \otimes (C_1 \oplus C_2) = C_1$;*
G_5. *for each $\vec{\Lambda} \in \Lambda^\star$ and $C \in \mathcal{D}$, it is $FV\big(\bar{\exists}_{\vec{\Lambda}} C\big) \subseteq \vec{\Lambda}$.*

A c.s. induces the relation $\vdash \subseteq \mathcal{D} \times \mathcal{D}$ given, for each $C_1, C_2 \in \mathcal{D}$, by

$$C_1 \vdash C_2 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad C_1 \otimes C_2 = C_1. \tag{2}$$

The relation '$\vdash$' is referred to as the approximation ordering *of the constraint system. The notation $C_1 \Vdash C_2$ stands for $C_1 \vdash C_2 \wedge C_1 \neq C_2$.*

In the sequel we will feel free to drop the quantifiers from the notation of the families of projection operators and diagonal elements. Condition G_4 can be restated as

$$C_1 \vdash C_1 \oplus C_2 \quad \text{and} \quad C_2 \vdash C_1 \oplus C_2.$$

[1] This definition of FV is an adaptation of the one of *dependent variables* given in [33, Definition 2.3].

In this form it clearly stands for the correctness of the merge operation, characterizing it as a (not necessarily least) upper bound operator with respect to the approximation ordering.

Hypothesis 7 *Indeed, constraint systems must satisfy some other (very technical) conditions related to how they deal with variables. For instance, they do not invent new free variables:* $FV(C_1 \otimes C_2) \subseteq FV(C_1) \cup FV(C_2)$, *and similarly for the merge operator. The operators are also* generic *in that they are insensible to variable names. This implies that, if* $[\bar{Y}/\bar{X}]$ *is a renaming for both* C_1 *and* C_2, *then*

$$(C_1 \otimes C_2)[\bar{Y}/\bar{X}] = C_1[\bar{Y}/\bar{X}] \otimes C_2[\bar{Y}/\bar{X}].$$

In particular, if we have also $C_1 \vdash C_2$, *then* $C_1[\bar{Y}/\bar{X}] \vdash C_2[\bar{Y}/\bar{X}]$. *In the sequel we will take all these overwhelmingly reasonable requirements for granted.*

Constraint systems enjoy several properties.

Proposition 8 (Properties of c.s.) *Any constraint system satisfies the following properties, for each* $C, C_1, C_2 \in \mathcal{D}$*:*

(i) $\langle \mathcal{D}, \vdash, \otimes, \mathbf{0}, \mathbf{1} \rangle$ *is a bounded meet-semilattice;*
(ii) $C \oplus \mathbf{1} = \mathbf{1}$*;*
(iii) $C_1 \vdash C_1 \oplus (C_1 \otimes C_2)$*;*
(iv) $C_1 \oplus C_2 = C_2 \quad \Longrightarrow \quad C_1 \vdash C_2$.

Observe that $\langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1} \rangle$, 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 (axiom G_4 of Definition 6), only one direction of the dual law is generally valid (property (iii) of Proposition 8). In particular, '$\oplus$' might be not component-wise monotone with respect to '$\vdash$', and '$\oplus$' does not distribute, in general, over '$\otimes$' (this would imply the equivalence of the two absorption laws).

So far for generic constraint systems, we consider now some strengthenings of Definition 6.

Definition 9 (Stronger c.s.'s) *Consider the following conditions:*

G_c. *for each family* $\{C_i \in \mathcal{D}\}_{i \in \mathbb{N}}$, $\bigoplus_{i \in \mathbb{N}} C_i \stackrel{\text{def}}{=} C_1 \oplus C_2 \oplus \cdots$ *exists and is unique in* $\mathcal{D}$*; moreover, associativity, commutativity, and idempotence of '*$\oplus$*' apply to denumerable as well as to finite families of operands;*
G_d. $\langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1} \rangle$ *is a distributive lattice;*
G_D. *for each* $C \in \mathcal{D}$ *and each family* $\{C_i \in \mathcal{D}\}_{i \in \mathbb{N}}$ *such that* $\bigoplus_{i \in \mathbb{N}} C_i$ *exists, we have* $C \otimes (\bigoplus_{i \in \mathbb{N}} C_i) = \bigoplus_{i \in \mathbb{N}} (C \otimes C_i)$*;*
G_N. *in* $\mathcal{D}$ *every strictly ascending chain,* $C_0 \Vdash C_1 \Vdash C_2 \Vdash \cdots$, *is finite.*

A constraint system is said closed *if it satisfies condition G_c. It is said* distributive *if it satisfies G_d. If it satisfies the stronger condition G_D then it is said* completely distributive. *Finally, a constraint system is said* Noetherian *if it satisfies the* ascending chain condition G_N.

So, the operation of merging together the information coming from all the computation paths always makes sense in a closed constraint system. Notice however that property G_c 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. Closedness will instead be a must for the constraint systems intended to capture "concrete" program semantics. Distributivity is useful for proving the equivalence of different abstract semantics constructions used for data-flow analysis. Complete distributivity is required for proving that a concrete semantics corresponds to the operational model of the language [21]. Observe that closed and completely distributive constraint systems are instances of the *closed semi-rings* used in [21].

For the abstract semantics constructions we will make use of another class of operators over constraints. These operators were introduced in [12].

Definition 10 (Widening.) [12] *Given a constraint system $\bar{\mathcal{D}}$, a binary operator $\nabla : \mathcal{D} \to \mathcal{D}$ is called a* widening for $\bar{\mathcal{D}}$ *if*

W_1. *for each $C_1, C_2 \in \mathcal{D}$ we have $C_1 \vdash C_1 \nabla C_2$ and $C_2 \vdash C_1 \nabla C_2$;*
W_2. *for each increasing chain $C_0 \vdash C_1 \vdash C_2 \vdash \cdots$, the sequence given by $C'_0 \stackrel{\text{def}}{=} C_0$ and, for $n \geq 1$, $C'_n \stackrel{\text{def}}{=} C'_{n-1} \nabla C_n$, is stationary after some $k \in \mathbb{N}$.*

Widenings allow to define *convergence acceleration methods* which ensure termination of the "abstract interpreter". However, even when termination is granted anyway (e.g., when the constraint system is Noetherian), these methods are often crucial for achieving *rapid* termination, that is, for obtaining usable data-flow analyzers. More sophisticated methods for convergence acceleration exist which employ also *narrowing* operators, and more complex widenings than the ones defined above (see [16,15]).

3.4 Generalized semantics

In a generalized semantics setting, the first thing to do is to provide atomic constraints with an interpretation on the chosen constraint system. Suppose that we are interested in deriving information about just two kind of program points: clause's entries and clause's successful exits. In a data-flow analysis setting (where this is often the case) that is to say that we want to derive

call-patterns and *success-patterns*. In other words, for each clause we want to derive properties of the constraint store which are valid

- whenever the clause is invoked (call-patterns);
- whenever a computation starting with the invocation of the clause terminates with success (success-patterns).

Observe that call-patterns depend on the ordering of atoms in the body of clauses and on the selection rule employed. By means of program transformations similar to the *magic* one [8,19] we can obtain the call-patterns of the original program (with respect to the selection rule employed) as success-patterns of the transformed one. These transformations, in fact, besides modifying the clauses of the original program, introduce new clauses that characterize the conditions under which the original clauses are invoked. In the transformed program the ordering of atoms in the clause's bodies is no longer important. Notice that the technique proposed in [19], while restricted to logic programs [2], is more sophisticated than usual transformation approaches, and allows not to loose the connection between call and success patterns. For these reasons we will consider only one kind of program points: clause's exits. Furthermore, in our domains the operation capturing constraint composition is associative, commutative, and idempotent. This means that we can assume without prejudice that all the clauses are of the form

$$p(\vec{\Lambda}) :- \{c_1, \ldots, c_n\} \,\square\, \{b_1, \ldots, b_k\},$$

where $\{c_1, \ldots, c_n\}$ is a set of atomic constraints, and $\{b_1, \ldots, b_k\}$ is a set of atoms. All the other restrictions imposed by Definition 3 must continue to hold. We now must associate a meaning to the finite sets of atomic constraints which occurs in clauses.

Definition 11 (Constraint interpretation.) *Given a language* C *of atomic constraints and a constraint system* $\bar{\mathcal{D}}$*, a* constraint interpretation of C in $\bar{\mathcal{D}}$ *is a computable function* $[\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}} : \wp_f(\mathsf{C}) \to \mathcal{D}$.

Then usually one considers, instead of the *syntactic* program P, its *semantic* version over the domain $\bar{\mathcal{D}}$, obtained by interpreting the atomic constraints of clauses through $[\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}}$. For $r = 1, \ldots, \# P$, we denote the r-th clause of P by $P[r]$.

Definition 12 (Generalized program.) *When* $\bar{\mathcal{D}}$ *is a constraint system, a* CLP($\bar{\mathcal{D}}$) program *is a sequence of Horn-like formulas of the form*

$$p(\vec{\Lambda}) :- C \,\square\, \{b_1, \ldots, b_k\},$$

[2] Actually, we have developed a technique, still based on program transformation, which is general enough to accommodate the entire CLP framework [37].

where $C \in \bar{\mathcal{D}}$ *is finitely representable. Given a* CLP(C) *program* P *and a constraint interpretation* $[\![\cdot]\!]^{\bar{\mathcal{D}}}_{\mathsf{C}}$*, the* CLP($\bar{\mathcal{D}}$) *program* $[\![P]\!]^{\bar{\mathcal{D}}}_{\mathsf{C}}$ *is given, for each* $r = 1, \ldots, \# P$*, by*

$$[\![P]\!]^{\bar{\mathcal{D}}}_{\mathsf{C}}[r] \equiv p(\vec{\Lambda}) :- [\![C]\!]^{\bar{\mathcal{D}}}_{\mathsf{C}} \square B \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad P[r] \equiv p(\vec{\Lambda}) :- C \square B. \tag{3}$$

An interpretation for a program P over a constraint system $\bar{\mathcal{D}}$ is a function from its program points (one for each clause) to $\mathcal{D}$.

Definition 13 (Interpretation.) *Let* $\bar{\mathcal{D}}$ *be a constraint system, and* P *a* CLP($\bar{\mathcal{D}}$) *program. An* interpretation for P over $\bar{\mathcal{D}}$ *is any element of*

$$\mathcal{I}^{\bar{\mathcal{D}}}_P \stackrel{\text{def}}{=} \{1, \ldots, \# P\} \to \mathcal{D}.$$

All the operations and relations over $\bar{\mathcal{D}}$ *are extended point-wise to* $\mathcal{I}^{\bar{\mathcal{D}}}_P$*. In particular* $\mathcal{I}^{\bar{\mathcal{D}}}_P$ *is partially ordered by the lifting of* $\vdash$*, i.e., for each* $I_1, I_2 \in \mathcal{I}^{\bar{\mathcal{D}}}_P$*,*

$$I_1 \vdash I_2 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall r \in \{1, \ldots, \# P\} : I_1(r) \vdash I_2(r).$$

We will represent interpretations by means of function graphs. It is straightforward to show that all the interesting properties of constraint systems lift smoothly to interpretations.

We are left with the choice of the semantics construction, that is, of the "interpretation transformer". For the purpose of this work we choose a bottom-up construction expressed by a flavor of the usual *immediate consequence operator* T_P, taking an interpretation and returning a new interpretation.

Our set of program variables is $\mathit{Vars} = \Lambda \uplus V$, where Λ and V are totally ordered. For $\vec{\Lambda} \in \Lambda^\star$ and $W \subseteq_f \mathit{Vars}$, we denote by $\bar{Y} \ll_{\vec{\Lambda}} W$ the fact that, with respect to the ordering of V, $\bar{Y}$ is a tuple of distinct consecutive variables in V such that $\# \bar{Y} = \# \vec{\Lambda}$ and the first element of $\bar{Y}$ immediately follows the greatest variable in W.

Definition 14 (Interpretation transformer.) *Let* P *be a* CLP($\bar{\mathcal{D}}$) *program, where* $\bar{\mathcal{D}}$ *is a constraint system. The operator induced by* P *over* $\mathcal{I}^{\bar{\mathcal{D}}}_P$*,* $T^{\bar{\mathcal{D}}}_P : \mathcal{I}^{\bar{\mathcal{D}}}_P \to \mathcal{I}^{\bar{\mathcal{D}}}_P$*, is*

$$T^{\bar{\mathcal{D}}}_P(I) \stackrel{\text{def}}{=} \left\{ \left(r, T^{\bar{\mathcal{D}}}_P(r, I)\right) \;\middle|\; 1 \le r \le \# P \right\}, \tag{4}$$

where $T_P^{\bar{\mathcal{D}}} \colon \{1, \ldots, \# P\} \times \mathcal{I}_P^{\bar{\mathcal{D}}} \to \mathcal{D}$ *is given by*

$$T_P^{\bar{\mathcal{D}}}(r, I) \stackrel{\text{def}}{=} \bigoplus \left\{ \bar{\exists}_{\vec{\Lambda}} \tilde{C} \;\middle|\; \begin{array}{l} P[r] \equiv p(\vec{\Lambda}) :- C \,\square\, \left\{ p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n) \right\} \\ \textit{and, for each } i = 1, \ldots, n\textit{:} \\ \qquad P[r_i] \equiv p_i(\vec{\Lambda}_i) :- C_i \,\square\, B_{r_i} \\ \qquad \bar{Y}_i \ll_{\vec{\Lambda}_i} FV\left(P[r]\right) \cup \bar{Y}_1 \cup \cdots \cup \bar{Y}_{i-1} \\ \qquad \tilde{C}_i = I(r_i)[\bar{Y}_i / \vec{\Lambda}_i] \\ \qquad C'_i = d_{\bar{X}_i \bar{Y}_i} \otimes \tilde{C}_i \\ \tilde{C} = C \otimes C'_1 \otimes \cdots \otimes C'_n \end{array} \right\}. \quad (5)$$

Notice that, in this construction, the merge operator is applied only to finite sets of operands. In summary, once we have fixed the constraint domain $\bar{\mathcal{D}}$ and the interpretation of atomic constraints $[\![\cdot]\!]_{\mathsf{C}}^{\mathcal{D}}$, the meaning of a CLP($\mathsf{C}$) program P over $\bar{\mathcal{D}}$ is encoded into the $T_P^{\bar{\mathcal{D}}}$ operator. Before rushing to require that $T_P^{\bar{\mathcal{D}}}$ must be continuous on the complete lattice $\bar{\mathcal{D}}$ we better have a closer look to our real needs.

3.5 Defining, computing and correlating non-standard semantics

Given our current focus on data-flow analysis of CLP programs, we only consider, for simplicity, the typical case in this field. On one hand we have a "concrete" constraint system $\bar{\mathcal{D}}^\natural$: it must capture the properties of interest, it must ensure the existence of the least fixpoint of $T_{P^\natural}^{\bar{\mathcal{D}}^\natural}$, that is, of *the* meaning of each program P. And, of course, this meaning must correspond to the one obtained by means of the top-down construction representing the operational model of the language (namely, some kind of extended SLD-resolution). This last requirement implies, as shown in [21], that $\bar{\mathcal{D}}^\natural$ must be closed and completely distributive, hence a complete lattice.

One the other hand, in data-flow analysis, we have an "abstract" constraint system $\bar{\mathcal{D}}^\sharp$. Here we are much less demanding: we simply want to compute in a *finite* way an approximation of a post-fixpoint of $T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp}$ (the least fixpoint might not even exist, or it might be too expensive to compute). And, of course, we need a guarantee that what we compute is a *correct* approximation of the concrete meaning. Finite computability can be ensured, in general, by using a widening operator. Thus, in this setting, the concrete and abstract iteration sequences defining, respectively, the concrete meaning and approximations of the abstract meaning of programs are quite different.

Definition 15 (Concrete and abstract iteration sequences.) *Consider a closed and completely distributive constraint system $\bar{\mathcal{D}}^\natural$, and a CLP($\bar{\mathcal{D}}^\natural$) program $P^\natural$. The* concrete iteration sequence *for $P^\natural$ is inductively defined as follows, for all ordinals $\kappa \in \mathbb{O}$:*

$$\begin{cases} T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow 0 \stackrel{\text{def}}{=} \mathbf{0}^\natural, \\ T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow (\kappa+1) \stackrel{\text{def}}{=} T_{P^\natural}^{\bar{\mathcal{D}}^\natural}\Big(T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow \kappa\Big), \\ T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow \kappa \stackrel{\text{def}}{=} \bigoplus_{\beta<\kappa}^\natural \Big(T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow \beta\Big), \quad \textit{when } \kappa > 0 \textit{ is a limit ordinal.} \end{cases} \tag{6}$$

Let $\bar{\mathcal{D}}^\sharp$ be any constraint system, and let $\nabla^\sharp$ be a widening operator over $\bar{\mathcal{D}}^\sharp$. For a CLP($\bar{\mathcal{D}}^\sharp$) *program $P^\sharp$, the* abstract iteration sequence for $P^\sharp$ with widening $\nabla^\sharp$ *is inductively defined by*

$$\begin{cases} T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \Uparrow 0 \stackrel{\text{def}}{=} \mathbf{0}^\sharp, \\ T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \Uparrow (k+1) \stackrel{\text{def}}{=} \Big(T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \Uparrow k\Big) \nabla^\sharp T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp}\Big(T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \Uparrow k\Big), \quad \textit{for } k \in \mathbb{N}. \end{cases} \tag{7}$$

Observe that, when $\bar{\mathcal{D}}^\sharp$ is Noetherian or when termination can be ensured in other ways, π_2 (the second projection) can be substituted for $\nabla^\sharp$ in (7). In these cases, indeed, the restriction of π_2 to the iterates' values is a widening operator. The following fact is easily proved using standard techniques [21].

Theorem 16 *If $\bar{\mathcal{D}}^\natural$ is a closed and completely distributive constraint system then, for each CLP($\bar{\mathcal{D}}^\natural$) program $P^\natural$ the least fixpoint of $T_{P^\natural}^{\bar{\mathcal{D}}^\natural}$ exists and is given by $lfp(T_{P^\natural}^{\bar{\mathcal{D}}^\natural}) = T_{P^\natural}^{\bar{\mathcal{D}}^\natural} \uparrow \omega$.*

We now come at the problem of ensuring the correctness of the analysis. We use an *abstraction correspondence* between the concrete and the abstract constraint systems, which induces an abstraction correspondence between the respective semantics [7,15].

Definition 17 (Abstraction function.) *Let $\bar{\mathcal{D}}^\natural$ and $\bar{\mathcal{D}}^\sharp$ be two constraint systems as in* Definition 15. *A function $\alpha : \mathcal{D}^\natural \to \mathcal{D}^\sharp$ is an* abstraction function *of $\bar{\mathcal{D}}^\natural$ into $\bar{\mathcal{D}}^\sharp$ if and only if*

A_1. α is a semi-morphism, *namely, for each $C^\natural, C_1^\natural, C_2^\natural, d_{\bar{X}\bar{Y}}^\natural \in \mathcal{D}^\natural$ and $\vec{\Lambda} \in \Lambda^\star$:*

$$\begin{array}{ll} \alpha(C_1^\natural \otimes^\natural C_2^\natural) \vdash^\sharp \alpha(C_1^\natural) \otimes^\sharp \alpha(C_2^\natural), & \alpha(\mathbf{0}^\natural) \vdash^\sharp \mathbf{0}^\sharp, \\ \alpha(C_1^\natural \oplus^\natural C_2^\natural) \vdash^\sharp \alpha(C_1^\natural) \oplus^\sharp \alpha(C_2^\natural), & \alpha\Big(d_{\bar{X}\bar{Y}}^\natural\Big) \vdash^\sharp d_{\bar{X}\bar{Y}}^\sharp, \\ \alpha\Big(\exists_{\vec{\Lambda}}^\natural C^\natural\Big) \vdash^\sharp \exists_{\vec{\Lambda}}^\sharp \alpha(C^\natural). & \end{array}$$

A_2. *for each increasing chain* $\{C_j^\sharp \in \mathcal{D}^\sharp\}_{j\in\mathbb{N}}$,

$$\exists C^\sharp \in \mathcal{D}^\sharp \,.\, \forall j \in \mathbb{N} : \alpha(C_j^\sharp) \vdash^\sharp C^\sharp \quad \Longrightarrow \quad \alpha\Big(\bigoplus_{j\in\mathbb{N}}^\sharp C_j^\sharp\Big) \vdash^\sharp C^\sharp;$$

A_3. *for each* $C^\sharp \in \mathcal{D}^\sharp$ *and each renaming* $[\bar{Y}/\bar{X}]$ *for* $C^\sharp$, *it happens that*

$$\alpha(C^\sharp)[\bar{Y}/\bar{X}] = \alpha\Big(C^\sharp[\bar{Y}/\bar{X}]\Big).$$

Any abstraction function $\alpha\colon \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *is naturally extended to* $\alpha\colon \mathcal{I}^{\bar{\mathcal{D}}^\sharp}_{P^\sharp} \to \mathcal{I}^{\bar{\mathcal{D}}^\sharp}_{P^\sharp}$, *when* $\#\, P^\sharp = \#\, P^\sharp$, *in the obvious way.*

As anticipated above, one of the beautiful things of the generalized approach is that the abstract meaning of CLP programs can be encoded into other CLP programs. We have thus an *abstract compilation* approach, where the soundness of the *compilation function* $[\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp}$ is expressed, for CLP(C) programs, by the requirement $\alpha \circ [\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp} \vdash^\sharp [\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp}$. The following result is an application of a theorem in [15, Proposition 6.20].

Theorem 18 *Given a* CLP(C) *program* P, *two constraint systems* $\bar{\mathcal{D}}^\sharp$ *and* $\bar{\mathcal{D}}^\sharp$, *the constraint interpretations* $[\![\cdot]\!]_{\mathsf{C}}^{\mathcal{D}^\sharp}$ *and* $[\![\cdot]\!]_{\mathsf{C}}^{\mathcal{D}^\sharp}$, *and the abstraction function* $\alpha\colon \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *such that* $\alpha \circ [\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp} \vdash^\sharp [\![\cdot]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp}$, $P^\sharp = [\![P]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp}$ *and* $P^\sharp = [\![P]\!]_{\mathsf{C}}^{\bar{\mathcal{D}}^\sharp}$, *we have that the abstract iteration with widening* (7) *is eventually stable after* $\ell \in \mathbb{N}$ *steps and*

$$\alpha\Big(\mathit{lfp}(T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp})\Big) = \alpha\Big(T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \uparrow \omega\Big) \vdash^\sharp \Big(T_{P^\sharp}^{\bar{\mathcal{D}}^\sharp} \Uparrow \ell\Big).$$

We now describe a hierarchy of constraint systems which capture most of the analysis domains used for deriving monotonic properties of programs, as well as the "concrete" collecting semantics they abstract. The basis is constituted by any constraint system which satisfies the conditions of Definition 6. In Section 5 we will show a way (which, of course, is not the only one) of defining such a structure. This requires introducing *simple constraint systems*, which is the purpose of the following section.

4 Simple constraint systems

A constraint system can be built starting from a set of finite constraints (or *tokens*), each expressing some partial information. We now define a notion of *simple constraint systems* (or s.c.s.), very similar to the one introduced in [35].

Definition 19 (Simple constraint system.) *A* simple constraint system *is a structure* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$, *where* $\mathcal{C}$ *is a set of constraints,* $\bot \in \mathcal{C}$, $\top \in \mathcal{C}$, *and*

$\vdash \subseteq \wp_f(\mathcal{C}) \times \mathcal{C}$ *is an* entailment relation *such that, for each* $C, C' \in \wp_f(\mathcal{C})$, *each* $c, c' \in \mathcal{C}$, *and* $X, Y \in Vars$*:*

$E_1.\ c \in C \quad \Longrightarrow \quad C \vdash c;$
$E_2.\ C \vdash \top;$
$E_3.\ (C \vdash c) \wedge (\forall c' \in C : C' \vdash c') \quad \Longrightarrow \quad C' \vdash c;$
$E_4.\ \{\bot\} \vdash c;$
$E_5.\ C \vdash c \quad \Longrightarrow \quad C[Y/X] \vdash c[Y/X].$

The '$\vdash$*' symbol is overloaded to denote also the extension* $\vdash \subseteq \wp(\mathcal{C}) \times \wp(\mathcal{C})$ *such that, for each* $C, C' \in \wp(\mathcal{C})$,

$$C \vdash C' \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall c' \in C' : \exists C'' \subseteq_f C \,.\, C'' \vdash c'.$$

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 8 and [36]). E_4 ensures that $\mathcal{C}$ is a finitely generable element (see Definition 21). Condition E_5, referred to as *genericity*, states that the entailment is insensible to variables' names [3].

By axioms E_1 and E_3 of Definition 19 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 20 (Elements.) [35] *The* elements *of an s.c.s.* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *are the* entailment-closed *subsets of* $\mathcal{C}$*, namely those* $C \subseteq \mathcal{C}$ *such that, whenever* $\exists C' \subseteq_f C \,.\, C' \vdash c$*, then* $c \in C$*. The set of elements of* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *is denoted by* $|\mathcal{C}|$*.*

The poset of elements is thus given by $\langle |\mathcal{C}|, \supseteq \rangle$. Notice that we deviate from [35] in that we order our constraint systems in the dual way.

Definition 21 (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* $\rho\colon \wp(\mathcal{C}) \to \wp(\mathcal{C})$ *given, for each* $C \subseteq \mathcal{C}$*, by*

$$\rho(C) \stackrel{\text{def}}{=} \{\, c \mid \exists C' \subseteq_f C \,.\, C' \vdash c \,\}.$$

[3] In [33] a stronger notion of genericity is used, namely $C[t/X] \vdash c[t/X]$ whenever $C \vdash c$, for any term t. This is too strong for our purposes: e.g. it would force us to treat non-linear numeric constraints in the same way as linear ones in CLP($\mathcal{R}$).

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.

In general, describing the "standard" semantics of a CLP($\mathcal{X}$) language is done as follows. Let T be the theory which corresponds to the domain $\mathcal{X}$ [26]. 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 in some disguise. For CLP($\mathbb{R}$) [4] the theory RCF of real closed fields would do the job. We see now some examples of simple constraint systems.

4.1 The atomic simple constraint system

This is probably the simplest useful example of s.c.s. The tokens include variable names. A variable name, when present in a constraint, expresses the fact that the variable has some (unspecified) property. For instance, being definitely bound to a *ground* value. In this case, X is just a shorthand for $ground(X)$. This s.c.s. is thus given by $\mathcal{C} \stackrel{\text{def}}{=} Vars \cup \{\bot, \top\}$ and by the smallest relation $\vdash \subseteq \wp_f(\mathcal{C}) \times \mathcal{C}$ satisfying conditions E_1–E_5 of Definition 19. In the sequel we will refer to this structure as to the *atomic* s.c.s.

A useful extension is to include tokens involving two variable names. These tokens state that the two variables involved share the property of interest: one enjoys it iff the other one does. More formally, we have

$$\mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C} \cup \{\, X \leftrightharpoons Y \mid X, Y \in Vars \,\},$$

and the entailment relation is suitably extended to $\mathcal{C}'$ requiring, for each $X, Y, Z \in Vars$:

$$\{X \leftrightharpoons Y\} \vdash Y \leftrightharpoons X; \quad \{X \leftrightharpoons Y, Y \leftrightharpoons Z\} \vdash X \leftrightharpoons Z; \quad \{X, X \leftrightharpoons Y\} \vdash Y.$$

[4] Beware not to confuse CLP($\mathbb{R}$), the *idealized* language over the reals [28], with CLP($\mathcal{R}$), the (far from ideal) implemented language and system [27].

The analysis described in [5,6,3] is based on constraint inference, a variant of constraint propagation [18]. This technique, developed in the field of artificial intelligence, has been applied to temporal and spatial reasoning [1,38].

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. Let $\mathsf{F} \subset \mathbb{R}$ a computable set of numbers, e.g. some family of rational numbers. The set of *boundaries* is given by $\mathsf{B} \stackrel{\text{def}}{=} \mathsf{F} \cup \{+\infty, -\infty\}$, and is ordered by the natural extension of the '$\leq$' relation. The set I of intervals is

$$\begin{aligned}\mathsf{I} \stackrel{\text{def}}{=} &\left\{ (b_1, b_2) \;\middle|\; b_1 \in \mathsf{B}, b_2 \in \mathsf{B} \right\} \cup \left\{ [\, b_1, b_2) \;\middle|\; b_1 \in \mathsf{F}, b_2 \in \mathsf{B} \right\} \\ &\cup \left\{ (b_1, b_2\,] \;\middle|\; b_1 \in \mathsf{B}, b_2 \in \mathsf{F} \right\} \cup \left\{ [\, b_1, b_2\,] \;\middle|\; b_1 \in \mathsf{F}, b_2 \in \mathsf{F} \right\}.\end{aligned}$$

Since any $I \in \mathsf{I}$ is a finite representations of a subset of $\mathbb{R}$ we will confuse intervals with their denotation. Thus we can state that intervals are closed under intersection, i.e., for each $I_1, I_2 \in \mathsf{I}$, $I_1 \cap I_2 \in \mathsf{I}$. The set of arithmetic relations is $\mathsf{R} \stackrel{\text{def}}{=} \{=, \neq, \leq <, \geq, >\}$ and our constraints are given by

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ e_1 \bowtie e_2 \;\middle|\; e_1, e_2 \in \mathsf{E}, \bowtie \in \mathsf{R} \right\} \cup \{\, e \lhd I \mid e \in \mathsf{E}, I \in \mathsf{I} \,\} \cup \{\bot, \top\}.$$

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

The approximate inference techniques we are interested in can be encoded into an entailment relation '$\vdash$' over $\mathcal{C}$. First we need to specify how we deal with intervals: we can intersect them, weaken them, and we detect failure by recognizing the empty ones:

$$\begin{aligned}\{e \lhd I_1, e \lhd I_2\} &\vdash e \lhd I_1 \cap I_2, \\ \{e \lhd I\} &\vdash e \lhd I', && \text{if } I \subseteq I', \\ \{e \lhd I\} &\vdash \bot, && \text{if } I = \varnothing.\end{aligned}$$

Two techniques for exploiting pure qualitative information are *symmetric* and *transitive closure*:

$$\begin{aligned}\{e_1 \bowtie e_2\} &\vdash e_2 \bowtie^{-1} e_1, \\ \{e_1 \bowtie e_2, e_2 \bowtie' e_3\} &\vdash e_1 \bowtie'' e_3, \qquad \text{if } \bowtie'' = tc(\bowtie, \bowtie'),\end{aligned}$$

where $\bowtie^{-1}$ is the inverse of $\bowtie$ (e.g. $<$ is the inverse of $>$, $\geq$ of $\leq$ and so on), and $tc\colon \mathsf{R} \times \mathsf{R} \rightarrowtail \mathsf{R}$ is the partial function individuated by the following table:

tc	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$<$	$<$	$<$			$<$	
$\leq$	$<$	$\leq$			$\leq$	
$>$			$>$	$>$	$>$	
$\geq$			$>$	$\geq$	$\geq$	
$=$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$\neq$					$\neq$	

This technique allows the inference of $A < C$ from $A \leq B$ and $B < C$. Of course, qualitative information can be combined and can led to the detection of inconsistencies:

$$\{e_1 \bowtie e_2, e_1 \bowtie' e_2\} \vdash e_1 \bowtie'' e_2, \quad \text{if } \forall x, y \in \mathbb{R} : (x \bowtie y \wedge x \bowtie' y) \Rightarrow x \bowtie'' y,$$

$$\{e_1 \bowtie e_2, e_1 \bowtie' e_2\} \vdash \bot, \quad \text{if } \forall x, y \in \mathbb{R} : \neg(x \bowtie y \wedge x \bowtie' y).$$

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 $e_1, \ldots, e_k$ 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 $\ddot{f}\colon \mathsf{I}^k \rightarrow \mathsf{I}$ is such that for each $x_1 \in I_1, \ldots, x_k \in I_k$, it happens that $f(x_1, \ldots, x_k) \in \ddot{f}(I_1, \ldots, I_k)$. For example, $A \lhd [\,3, 6) \wedge B \lhd [\,-1, 5\,] \vdash A + B \lhd [\,2, 11)$. 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

$$\{e_1 \lhd I_1, e_2 \lhd I_2\} \vdash e_1 \bowtie e_2, \quad \text{if } \forall x_1 \in I_1, x_2 \in I_2 : x_1 \bowtie x_2.$$

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 $I_1' \stackrel{\text{def}}{=} \{\, x_1 \in I_1 \mid \exists x_2 \in I_2 \;.\; x_1 \bowtie x_2 \,\}$. This is an example of *local-consistency technique* [32,30]. 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 techniques. Other techniques, such as *interval refinement*, can be easily incorporated. What we have just presented is a watered-down version of the

numerical component (presented as a simple constraint system) employed in the CHINA analyzer [3].

5 Determinate constraint systems

Determinate constraint systems are at the bottom of the hierarchy. Such a construction is uniquely determined by a simple constraint system together with appropriate merge operator and diagonal elements. Notice that, for simplicity, we only present the *finite fragment* of the constraint system, that is, the sub-structure consisting of the finite elements only.

Definition 22 (Determinate constraint system.) *Let* $\mathcal{S} \stackrel{\text{def}}{=} \langle \mathcal{C}, \vdash, \bot, \top \rangle$ *be a simple constraint system. Let* $\mathbf{0}, \mathbf{1} \in |\mathcal{C}|_0$ *and* $\otimes \colon |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *be given, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, by*

$$\mathbf{0} \stackrel{\text{def}}{=} \mathcal{C}, \qquad \mathbf{1} \stackrel{\text{def}}{=} \rho(\varnothing), \qquad C_1 \otimes C_2 \stackrel{\text{def}}{=} \rho(C_1 \cup C_2).$$

Let $\oplus \colon |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *be an operator satisfying conditions* G_2 *and* G_4 *of* Definition 6. *The projection operators* $\bar{\exists}_{\vec{\Lambda}} \colon |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *are given, for each* $\vec{\Lambda} \in \Lambda^\star$ *and each* $C \in |\mathcal{C}|_0$*, by*

$$\bar{\exists}_{\vec{\Lambda}} C \stackrel{\text{def}}{=} \rho\Big(\Big\{ c \in C \;\Big|\; FV(c) \subseteq \vec{\Lambda} \Big\}\Big).$$

Finally, let $\{d_{\bar{X}\bar{Y}}\}_{\bar{X},\bar{Y} \in Vars^\star}$ *a family of elements of* $|\mathcal{C}|_0$*. We will call the structure* $\langle |\mathcal{C}|_0, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{\Lambda}}\}, \{d_{\bar{X}\bar{Y}}\} \rangle$ *a* determinate constraint system *over* $\mathcal{S}$ *and '*$\oplus$*'.*

Theorem 23 *Each determinate constraint system is indeed a constraint system. Also, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, we have* $C_1 \vdash C_2 \iff C_1 \supseteq C_2$*.*

The choice of a suitable merge operator, required in addition to an s.c.s. to obtain a determinate constraint system, can be done with relative freedom. This freedom can often be conveniently exploited in order to get a reasonable complexity/precision tradeoff. The same will apply to the *ask-and-tell constraint systems* of Section 7.

Observe that, given $C_1, C_2 \in |\mathcal{C}|_0$, there is no *a priori* guarantee that $C = C_1 \cap C_2 \in |\mathcal{C}|_0$. In fact, there are simple constraint systems where this is false. Defining the merge operator as set intersection, however, works in many cases.

A trivial example of merge operator is the following, whose definition is independent from the simple constraint system at hand:

$$C_1 \oplus C_2 \stackrel{\text{def}}{=} \begin{cases} C_1, & \text{if } C_1 = C_2 \text{ or } C_2 = \mathbf{0}; \\ C_2, & \text{if } C_1 = \mathbf{0}; \\ \mathbf{1}, & \text{otherwise.} \end{cases} \tag{8}$$

For a less trivial example, suppose we are approximating subsets of $\mathbb{R}^n$ by means of (closed) convex polyhedra. Of course they will be represented by sets of linear disequations over $x_1, \ldots, x_n$, but, for the purpose of the present example, we will consider the polyhedra themselves. For any convex polyhedra $X, Y \subseteq \mathbb{R}^n$, define $X \vdash Y$ iff $X \subseteq Y$ and

$$X \oplus Y \stackrel{\text{def}}{=} \begin{cases} X, & \text{if } X = Y \text{ or } Y = \varnothing; \\ Y, & \text{if } X = \varnothing; \\ \mathrm{bb}(X \cup Y), & \text{otherwise,} \end{cases} \tag{9}$$

where $\mathrm{bb}(Z)$ is the smallest "bounding box" containing $Z \subseteq \mathbb{R}^n$, namely

$$\mathrm{bb}(Z) \stackrel{\text{def}}{=} \Big\{ (x_1, \ldots, x_n) \;\Big|\; \forall i = 1, \ldots, n : \inf \pi_i(Z) \leq x_i \leq \sup \pi_i(Z) \Big\}.$$

The most precise merge operator is, of course, given by the convex hull, i.e.

$$X \oplus Y \stackrel{\text{def}}{=} \min\Big\{ W \subseteq \mathbb{R}^n \;\Big|\; W \supseteq X \cup Y \text{ and } W \text{ is a c.p.} \Big\}. \tag{10}$$

Notice that (10) satisfies both the absorption laws (thus giving rise to a lattice), (8) and (9) do not. None of them results in a distributive constraint system. Furthermore, (8) and (9) are closed, while (10) is not.

5.1 Definiteness analysis: Con

Consider the extension of the *atomic* simple constraint system, $\mathcal{C}'$, introduced in Section 4.1, and apply to it the determinate constraint system construction with $C_1 \oplus C_2 \stackrel{\text{def}}{=} C_1 \cap C_2$ for each $C_1, C_2 \in |\mathcal{C}'|$. Let also the diagonal elements be given, for each $\bar{X}, \bar{Y} \in \mathit{Vars}^\star$ of the same cardinality, by

$$d_{\bar{X}\bar{Y}} \stackrel{\text{def}}{=} \rho\Big(\{\, \pi_i(\bar{X}) \leftrightharpoons \pi_i(\bar{Y}) \mid 1 \leq i \leq \#\,\bar{X} \,\} \Big).$$

The resulting domain (a closed and Noetherian d.c.s.) is the simplest one for definiteness analysis, and it was used in early groundness analyzers [31,29]. The name *Con* comes from the fact that elements of the form $\{X_1, \ldots, X_n\}$

are usually regarded as the conjunction $X_1 \wedge \cdots \wedge X_n$, meaning that $X_1, \ldots, X_n$ are definitely bound to a unique value. In this view '$\otimes$' corresponds to logical conjunction. *Con* is a very weak domain for definiteness analysis. It cannot capture neither "aliasing" (apart from the special kind of aliasing arising from parameter passing) nor more complex dependencies between variables such as those implied by "concrete" constraints like $A = f(B, C)$ and $A + B + C = 0$. Moreover it cannot represent or exploit disjunctive information.

6 Powerset constraint systems

For the purpose of program analysis of monotonic properties 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 [15]. For doing that we need some notions from the theory of posets.

Given a poset $\langle L, \bot, \leq \rangle$, the relation $\preceq \subseteq \wp(L) \times \wp(L)$ induced by $\leq$ is given, for each $S_1, S_2 \in \wp(L)$ by

$$(S_1 \preceq S_2) \iff (\forall x_1 \in S_1 : \exists x_2 \in S_2 \,.\, x_1 \leq x_2).$$

A subset $S \in \wp(L)$ is said *non-redundant* iff $\bot \notin S$ and

$$\forall x_1, x_2 \in S : x_1 \leq x_2 \implies x_1 = x_2.$$

The set of non-redundant subsets of L wrt $\leq$ is denoted by $\wp_n(L, \leq)$. The function $\Omega^{\leq}_L : \wp(L) \to \wp_n(L, \leq)$ mapping each set into its non-redundant counterpart is given, for each $S \in \wp(L)$, by

$$\Omega^{\leq}_L(S) = S \setminus \{\, x \in S \mid x = \bot \vee \exists x' \in S \,.\, x < x' \,\}.$$

Thus, for $S \in \wp(L)$, $\Omega^{\leq}_L(S)$ is the set of maximal elements of S. However, there is no guarantee, in general, that such maximal elements exist: L could be an infinite chain without an upper bound in L, and thus would be mapped to $\varnothing$ by $\Omega^{\leq}_L$. We will denote by $\wp_c(L)$ the set of all those $S \in \wp(L)$ such that, if S contains an infinite chain C, then it also contains an upper bound for C. Observe that $\wp_f(L) \subseteq \wp_c(L)$ and that, if L satisfies the ascending chain condition, then $\wp(L) = \wp_c(L)$.

The powerset construction upgrades a domain by considering sets of elements of the base-level domain which are non-redundant with respect to the approximation ordering.

Definition 24 (Powerset constraint systems.) *Given a Noetherian constraint system* $\bar{\mathcal{D}} = \langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{\Lambda}}\}, \{d_{\bar{X}\bar{Y}}\}\rangle$*, the* powerset constraint system *over* $\bar{\mathcal{D}}$ *is given by*

$$\Big\langle \wp_n(\mathcal{D}, \vdash), \otimes_{\mathrm{P}}, \oplus_{\mathrm{P}}, \mathbf{0}_{\mathrm{P}}, \mathbf{1}_{\mathrm{P}}\{\bar{\exists}^{\mathrm{P}}_{\vec{\Lambda}}\}, \{d^{\mathrm{P}}_{\bar{X}\bar{Y}}\}\Big\rangle,$$

where

$$\begin{aligned}
S_1 \otimes_{\mathrm{P}} S_2 &\stackrel{\mathrm{def}}{=} \Omega^{\vdash}_{\mathcal{D}}\Big(\{\, C_1 \otimes C_2 \mid C_1 \in S_1, C_2 \in S_2 \,\}\Big),\\
S_1 \oplus_{\mathrm{P}} S_2 &\stackrel{\mathrm{def}}{=} \Omega^{\vdash}_{\mathcal{D}}(S_1 \cup S_2),\\
\mathbf{0}_{\mathrm{P}} &\stackrel{\mathrm{def}}{=} \varnothing,\\
\mathbf{1}_{\mathrm{P}} &\stackrel{\mathrm{def}}{=} \{\mathbf{1}\},\\
\bar{\exists}^{\mathrm{P}}_{\vec{\Lambda}}\, S &\stackrel{\mathrm{def}}{=} \Omega^{\vdash}_{\mathcal{D}}\Big(\{\, \bar{\exists}_{\vec{\Lambda}}\, C \mid C \in S \,\}\Big),\\
d^{\mathrm{P}}_{\bar{X}\bar{Y}} &\stackrel{\mathrm{def}}{=} \{d_{\bar{X}\bar{Y}}\}.
\end{aligned}$$

If $\bar{\mathcal{D}}$ *is* any *constraint system, the* finite powerset constraint system *over* $\bar{\mathcal{D}}$ *is*

$$\Big\langle \wp_n(\mathcal{D}, \vdash) \cap \wp_f(\mathcal{D}), \otimes_{\mathrm{P}}, \oplus_{\mathrm{P}}, \mathbf{0}_{\mathrm{P}}, \mathbf{1}_{\mathrm{P}}\{\bar{\exists}^{\mathrm{P}}_{\vec{\Lambda}}\}, \{d^{\mathrm{P}}_{\bar{X}\bar{Y}}\}\Big\rangle,$$

where all the operators are as above.

This double definition reflects the two possible uses of powerset constraint systems. One is to define concrete domains in those cases where the base-level constraint system is Noetherian. The other is when designing abstract domains and, clearly, only the finite elements are of interest. In both cases, when we deal with monotonic properties, we loose nothing if we restrict ourselves to non-redundant sets in order to capture the non-determinism of CLP languages. This is a consequence of the fact that, when $\langle L, \bot, \leq\rangle$ is a poset, we have both $\Omega^{\leq}_{L}(S) \preceq S$ and $S \preceq \Omega^{\leq}_{L}(S)$, for each $S \in \wp_c(L)$. Of course, when the base-level c.s. is not Noetherian, one has to consider all the subsets in the design of a concrete domain.

Theorem 25 *Any powerset constraint system built over a Noetherian c.s.* $\bar{\mathcal{D}}$*,*

$$\Big\langle \wp_n(\mathcal{D}, \vdash), \otimes_{\mathrm{P}}, \oplus_{\mathrm{P}}, \mathbf{0}_{\mathrm{P}}, \mathbf{1}_{\mathrm{P}}\{\bar{\exists}^{\mathrm{P}}_{\vec{\Lambda}}\}, \{d^{\mathrm{P}}_{\bar{X}\bar{Y}}\}\Big\rangle,$$

is a closed and completely distributive constraint system, where the ordering is given, for each $S_1, S_2 \in \wp_n(\mathcal{D}, \vdash)$*, by*

$$S_1 \vdash_{\mathrm{P}} S_2 \iff \forall C_1 \in S_1 : \exists C_2 \in S_2 \,.\, C_1 \vdash C_2. \tag{11}$$

For any c.s. $\bar{\mathcal{D}}$*, the finite powerset c.s. built over* $\bar{\mathcal{D}}$ *is a distributive constraint system, where the ordering is given by* (11)*, for* $S_1, S_2 \in \wp_n(\mathcal{D}, \vdash) \cap \wp_f(\mathcal{D})$*.*

7 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 [34] similar constructions are called *ask-and-tell constraint systems*. In our construction, ask-and-tell constraint systems are built from constraint systems by regarding some kernel operators as constraints. We follow [34] 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 [35] by means of a declarative semantics. Here we give also an operational characterization which is better suited to our needs.

Definition 26 (Finite cc agents: syntax.) *A* finite cc agent *over a constraint system* $\bar{\mathcal{D}} = \langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{X}}\}, \{d_{\bar{X}\bar{Y}}\}\rangle$ *is any string generated by the following grammar:*

$$\mathbf{A} ::= \mathrm{tell}(C) \mid \mathrm{ask}(C) \to \mathbf{A} \mid \mathbf{A} \parallel \mathbf{A}$$

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

$$\mathrm{ask}(C_1; \ldots ; C_n) \to \mathbf{A} \quad \equiv \quad \Big(\mathrm{ask}(C_1) \to \mathbf{A}\Big) \parallel \cdots \parallel \Big(\mathrm{ask}(C_n) \to \mathbf{A}\Big).$$

When this will not cause confusion we will freely drop the syntactic sugar, writing C and $C \to A$ where $\mathrm{tell}(C)$ and $\mathrm{ask}(C) \to A$ are intended. One of the beautiful properties of kernel operators is that they can be uniquely represented by their range, i.e. the set of their fixed points [23]. The denotational semantics of finite cc agents over $\bar{\mathcal{D}}$ is thus conveniently expressed by a function $[\![\cdot]\!] \colon \mathcal{A}(\bar{\mathcal{D}}) \to \wp(\mathcal{D})$ defined following [35].

Definition 27 (Semantics of finite cc agents.) *The semantics of finite* cc *agents is given by the following equations:*

$$[\![C]\!] \stackrel{\mathrm{def}}{=} {\downarrow}C, \qquad [\![C \to A]\!] \stackrel{\mathrm{def}}{=} \overline{{\downarrow}C} \cup {\downarrow}A, \qquad [\![A \parallel B]\!] \stackrel{\mathrm{def}}{=} [\![A]\!] \cap [\![B]\!].$$

Observe that the actual kernel operator A^K corresponding to a finite agent $A \in \mathcal{A}(\bar{\mathcal{D}})$ can be recovered from $[\![A]\!]$ as $A^K \stackrel{\mathrm{def}}{=} \lambda C \,.\, \sup(({\downarrow}C) \cap [\![A]\!])$. The introduction of a syntactic normal form for finite cc agents allows to simplify the subsequent semantic treatment.

Definition 28 (Finite cc agents: syntactic normal form.) *The transformation η over $\mathcal{A}(\bar{\mathcal{D}})$ is defined, for $C^a, C_1^a, C_2^a, C^t \in \mathcal{D}$ and $A, A_1, A_2 \in \mathcal{A}(\bar{\mathcal{D}})$, as follows:*

$$\eta(C^a \to C^t) \stackrel{\text{def}}{=} \begin{cases} \mathbf{1} \to \mathbf{1}, & \text{if } C^a \vdash C^t; \\ C^a \to (C^a \otimes C^t), & \text{otherwise;} \end{cases}$$

$$\eta(C^t) \stackrel{\text{def}}{=} \mathbf{1} \to C^t;$$

$$\eta\Big(C_1^a \to (C_2^a \to A)\Big) \stackrel{\text{def}}{=} \eta\Big((C_1^a \otimes C_2^a) \to A\Big);$$

$$\eta\Big(C^a \to (A_1 \parallel A_2)\Big) \stackrel{\text{def}}{=} \eta\Big((C^a \to A_1) \parallel (C^a \to A_2)\Big);$$

$$\eta(A_1 \parallel A_2) \stackrel{\text{def}}{=} \eta(A_1) \parallel \eta(A_2).$$

The following fact is easily proved.

Proposition 29 *The transformation η of* Definition 28 *is well defined. Furthermore, if $A \in \mathcal{A}(\bar{\mathcal{D}})$ then $[\![\eta(A)]\!] = [\![A]\!]$ and $\eta(A)$ is of the form*

$$(C_1^a \to C_1^t) \parallel \cdots \parallel (C_n^a \to C_n^t), \quad \text{where } C_i^t \Vdash C_i^a \text{ for each } i = 1, \ldots, n.$$

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 syntactic 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. The semantics given in Definition 27 clearly allows that. From now on we will treat the '$\parallel$' operator as a (polymorphic) constructor for flat sets.

Definition 30 (A calculus of finite cc agents.) *Let $\mathbf{1}_{\mathrm{A}} \stackrel{\text{def}}{=} \mathbf{1} \to \mathbf{1}$. The* structural congruence *of the calculus is the smallest congruence relation $\equiv_s$ such that $\langle \mathcal{A}(\bar{\mathcal{D}}), \parallel, \mathbf{1}_{\mathrm{A}} \rangle_{/\equiv_s}$ is a commutative and idempotent monoid. The reduction rules of the calculus are given in* Figure 1. *We also define the relation $\rho_{\mathrm{A}} \subseteq \mathcal{A}(\bar{\mathcal{D}}) \times \mathcal{A}(\bar{\mathcal{D}})$ given, for each $A, A' \in \mathcal{A}(\bar{\mathcal{D}})$, by*

$$A \;\rho_{\mathrm{A}}\; A' \stackrel{\text{def}}{\Longleftrightarrow} \exists n \in \mathbb{N} \,.\, A = A_1 \wedge A_n = A' \wedge A_1 \mapsto A_2 \mapsto \cdots \mapsto A_n \mapsto$$

In the following we will systematically abuse the notation denoting $\mathcal{A}(\bar{\mathcal{D}})_{/\equiv_s}$ simply by $\mathcal{A}(\bar{\mathcal{D}})$. Consequently, every assertion concerning $\mathcal{A}(\bar{\mathcal{D}})$ is to be intended *modulo structural congruence*. We introduce now, following [35], a normal form for finite cc agents.

Structure

$$\frac{A_1 \equiv_s A_1' \quad A_1' \longmapsto A_2' \quad A_2' \equiv_s A_2}{A_1 \longmapsto A_2} \qquad \frac{A_1 \longmapsto A_1'}{A_1 \parallel A_2 \longmapsto A_1' \parallel A_2}$$

Reduction $r(1,2)$

$$\frac{C_1^a \vdash C_2^a \qquad C_1^a \otimes C_2^t \vdash C_1^t}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto (C_2^a \to C_2^t)}$$

Deduction $d(1,2)$

$$\frac{C_1^t \vdash C_2^a \qquad C_1^t \nvdash C_2^t}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto \left(C_1^a \to (C_1^t \otimes C_2^t)\right) \parallel (C_2^a \to C_2^t)}$$

Absorption $a(1,2)$

$$\frac{C_1^a \Vdash C_2^a \qquad C_1^a \nvdash C_2^t \qquad C_1^t \Vdash C_1^a \otimes C_2^t}{(C_1^a \to C_1^t) \parallel (C_2^a \to C_2^t) \longmapsto \left((C_1^a \otimes C_2^t) \to C_1^t\right) \parallel (C_2^a \to C_2^t)}$$

Fig. 1. Reduction rules for finite cc agents.

Definition 31 (Semantic normal form.) [35] *An agent* $A \in \mathcal{A}(\bar{\mathcal{D}})$ *is* in semantic normal form *if and only if* $A = \mathbf{1}_{\mathrm{A}}$ *or* $A = \|_{i=1}^{n} C_i^a \to C_i^t$ *and, for each* $i, j \in \{1, \ldots, n\}$*:*

N_1. $C_i^t \Vdash C_i^a$;
N_2. $i \neq j \implies C_i^a \neq C_j^a$;
N_3. $C_i^a \Vdash C_j^a \implies C_i^a \Vdash C_j^t$;
N_4. $C_i^t \vdash C_j^a \implies C_i^t \vdash C_j^t$.

It turns out that this normal form is indeed very strong, whence its name.

Theorem 32 [35] *Two agents* $A_1, A_2 \in \mathcal{A}(\bar{\mathcal{D}})$ *have the same semantic normal form if and only if* $[\![A_1]\!] = [\![A2]\!]$.

The purpose of our rewriting system is to put finite cc agents into semantic normal form, preserving their original semantics.

Theorem 33 *For each agent* $A \in \mathcal{A}(\bar{\mathcal{D}})$ *in syntactic normal form, if* $A \ \rho_{\mathrm{A}} \ A'$ *then* $[\![A]\!] = [\![A']\!]$ *and* A' *is in semantic normal form. The term-rewriting system depicted in* Figure 1 *is strongly normalizing. Thus the relation* ρ_{A} *is indeed a function* $\rho_{\mathrm{A}} \colon \mathcal{A}(\bar{\mathcal{D}}) \to \mathcal{A}(\bar{\mathcal{D}})$*. Finally, for* $A_1, A_2 \in \mathcal{A}(\bar{\mathcal{D}})$ *we have* $\rho_{\mathrm{A}}(A_1) = \rho_{\mathrm{A}}(A_2)$ *if and only if* $[\![A_1]\!] = [\![A2]\!]$.

The situation here is almost identical to the one of Definition 21, 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 34 (Elements.) *The* elements *of* $\mathcal{A}(\bar{\mathcal{D}})$ *are the fixed points of the* inference map ρ_{A}. *The set of elements of* $\mathcal{A}(\bar{\mathcal{D}})$ *will be denoted by* $|\mathcal{A}(\bar{\mathcal{D}})|$.

We are now in position to introduce a new class in our hierarchy of constraint systems.

Definition 35 (Ask-and-tell constraint system.) *Given a constraint system* $\bar{\mathcal{D}} = \langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\vec{\Lambda}}\}, \{d_{\bar{X}\bar{Y}}\}\rangle$, *let* $\mathcal{A} = |\mathcal{A}(\bar{\mathcal{D}})|$. *Then let* $\mathbf{0}_{\mathrm{A}}, \mathbf{1}_{\mathrm{A}} \in \mathcal{A}$, *and* $\otimes_{\mathrm{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ *be given, for each* $A_1, A_2 \in \mathcal{A}$, *by*

$$\mathbf{0}_{\mathrm{A}} \stackrel{\mathrm{def}}{=} \mathbf{1} \to \mathbf{0}, \qquad A_1 \otimes_{\mathrm{A}} A_2 \stackrel{\mathrm{def}}{=} \rho_{\mathrm{A}}(A_1 \parallel A_2).$$
$$\mathbf{1}_{\mathrm{A}} \stackrel{\mathrm{def}}{=} \mathbf{1} \to \mathbf{1},$$

The projection operators $\bar{\exists}^{\mathrm{A}}_{\vec{\Lambda}} : \mathcal{A} \to \mathcal{A}$ *are given, for each* $\vec{\Lambda} \subseteq_f Vars$ *and* $A \in \mathcal{A}$, *by*

$$\bar{\exists}^{\mathrm{A}}_{\vec{\Lambda}} A \stackrel{\mathrm{def}}{=} \rho_{\mathrm{A}} \left(\left\{ \left(\bar{\exists}_{\vec{\Lambda}} C^a \to \bar{\exists}_{\vec{\Lambda}} C^t \right) \;\middle|\; \begin{array}{l} \left(C^t \to C^a\right) \in A \quad \textit{and} \\ \left(\left(\mathbf{1} \to \bar{\exists}_{\vec{\Lambda}} C^a \right) \otimes_{\mathrm{A}} A \right) \vdash_{\mathrm{A}} \left(\mathbf{1} \to C^a \right) \end{array} \right\} \right).$$

Finally, let $\oplus_{\mathrm{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ *be an operator satisfying the conditions* G_2 *and* G_4 *of* Definition 6. *For any indexed family* $\{d^{\mathrm{A}}_{\bar{X}\bar{Y}}\}_{\bar{X},\bar{Y} \in Vars^\star}$ *of elements of* $\mathcal{A}$, *we will call* $\langle \mathcal{A}(\bar{\mathcal{D}}), \otimes_{\mathrm{A}}, \oplus_{\mathrm{A}}, \mathbf{0}_{\mathrm{A}}, \mathbf{1}_{\mathrm{A}}, \{\bar{\exists}^{\mathrm{A}}_{\vec{\Lambda}}\}, \{d^{\mathrm{A}}_{\bar{X}\bar{Y}}\}\rangle$ *an* ask-and-tell constraint system *over* $\bar{\mathcal{D}}$ *and* '$\oplus_{\mathrm{A}}$'.

Notice that, as far as the diagonal elements are concerned, we have left complete freedom. This is because, in an ask-and-tell construction, the induced diagonals $d^{\mathrm{A}}_{\bar{X}\bar{Y}} = \mathbf{1} \to d_{\bar{X}\bar{Y}}$ are not necessarily a good choice (see Section 7.3 for a simple example).

Theorem 36 *If* $\bar{\mathcal{D}}$ *is a c.s., then* $\langle |\mathcal{A}(\bar{\mathcal{D}})|, \otimes_{\mathrm{A}}, \oplus_{\mathrm{A}}, \mathbf{0}_{\mathrm{A}}, \mathbf{1}_{\mathrm{A}}, \{\bar{\exists}^{\mathrm{A}}_{\vec{\Lambda}}\}, d^{\mathrm{A}}_{\bar{X}\bar{Y}}\rangle$ *is so.*

The projection operators are indeed quite complicated. The problem is originated from requirement N_3 of the normal form of Definition 31. This requirement enforces the need of the absorption rule in the calculus. The rule, by strengthening the ask-constraint of pairs, introduces "false dependencies". Consider, for instance, a constraint system where elements include the finite subsets of $\{\, p(X), q(X), r(X) \mid X \in Vars \,\}$ and the operators $\otimes$ and $\oplus$ are set union and intersection, respectively. The non-normalized cc agent over this constraint system

$$A = \mathbf{1} \to \{q(Y)\} \parallel \{p(X)\} \to \{r(X), q(Y)\}$$

is normalized, by means of the absorption rule, to

$$A' = \mathbf{1} \to \{q(Y)\} \parallel \{p(X), q(Y)\} \to \{r(X), q(Y)\}.$$

The absorption rule has thus introduced the dependency of $r(X)$ from $q(Y)$, which is indeed false in the context of A' (as it was in the context of A). A definition of the projection operators not taking into account this phenomenon would cause the inaccurate result $\bar{\exists}^{\mathrm{A}}_X A' = \mathbf{1}_{\mathrm{A}}$. The projection operators given in Definition 35, instead, recognize the false dependency by noting that $\{p(X)\} = \bar{\exists}_X\{p(X), q(Y)\}$ is, in the context of A', equivalent to $\{p(X), q(Y)\}$, that is

$$\mathbf{1} \to \{p(X)\} \parallel A' \quad \vdash_{\mathrm{A}} \quad \mathbf{1} \to \{p(X), q(Y)\}.$$

We can thus obtain the expected result $\bar{\exists}^{\mathrm{A}}_X A' = \{p(X)\} \to \{r(X)\}$. We will see in a moment other problems provoked by the absorption rule and, in turn, by the normal form we employ for agents.

7.1 Merge operators

Even though the ask-and-tell construction is parameterized with respect to a merge operator, it is possible to induce such an operator from the one of the base-level constraint system. Since this is a problematic point we proceed with care.

Suppose that the base-level constraint system $\bar{\mathcal{D}}$ is a lattice. Thus kernel operators over $\bar{\mathcal{D}}$ form again a lattice, where the *lub* is given, for k_1 and k_2 kernel operators and for each $C \in \mathcal{D}$, by

$$(k_1 \sqcup k_2)(C) \stackrel{\text{def}}{=} k_1(C) \oplus k_2(C), \quad \text{for } C \in \mathcal{D}, \tag{12}$$

whose fixed points are

$$(k_1 \sqcup k_2)(\mathcal{D}) = \Big\{ C_1 \oplus C_2 \;\Big|\; C_1, C_2 \in k_1(\mathcal{D}) \cup k_2(\mathcal{D}) \Big\}.$$

In terms of kernel operators, as pointed out in [34], this can be thought of as a kind of *determinate disjunction*: $k_1 \sqcup k_2$ gives, on any input C, the strongest common information between k_1 and k_2. The computational significance of this concept has been first recognized in [39], where determinate disjunction allows for significant improvements in some constraint propagation algorithms.

The problem is that, even when k_1 and k_2 are represented by finite cc agents A_1 and A_2, namely $k_1 = A_1^K$ and $k_2 = A_2^K$, we have no guarantees whatsoever that $k_1 \sqcup k_2$ is representable by a finite cc agent. In other words, (syntactic) finite cc agents are not, in general, closed under the (semantic) *lub* operation.

As a consequence, we must content ourselves with upper bounds (unless we are willing to enrich our representation language with a construct like $A_1 + A_2$ expressing determinate disjunction, and we are not). Observe that the very precise effect of (12) can be obtained (at a consequently high cost) applying a powerset construction (Section 6) to the ask-and-tell constraint system considered. This way, when merging two (non-redundant) agents we will keep both of them, thus realizing, in practice, the '+' construct mentioned above. If we do that, obviously, there is no need at all to define a merge operator at the ask-and-tell level.

In our general situation, the base-level constraint system $\bar{\mathcal{D}}$ might not be a lattice, and (12) might not define a kernel operator. In these cases, an upper bound on the poset of kernel operators over $\bar{\mathcal{D}}$ can be given as

$$(k_1 \tilde{\sqcup} k_2)(C) \stackrel{\text{def}}{=} C \otimes \Big(k_1(C) \oplus k_2(C)\Big), \quad \text{for } C \in \mathcal{D}, \tag{13}$$

which, still, is not guaranteed to correspond to any finite cc agent over $\bar{\mathcal{D}}$. We stress again that our non-commitment to lattices in the general definition of constraint systems (Section 3.3) is not merely dictated by the desire of freely managing the complexity/precision tradeoff. In cases like the one at hand we have no other sensible choice due to representation problems.

Our study of computable merge operators starts with a simple operation merging two (not necessarily normalized) agents into one. This is done, roughly speaking, by taking the meet of the ask constraints, and the merge of the tell constraints.

Definition 37 (Merge operator over agents.) *Consider a constraint system $\bar{\mathcal{D}} = \langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\bar{\Lambda}}\}, \{d_{\bar{X}\bar{Y}}\}\rangle$, and any two finite* cc *agents over $\bar{\mathcal{D}}$ in syntactic normal form: $A_1 = \|_{i=1}^{n} C_i^a \to C_i^t$ and $A_2 = \|_{j=1}^{m} D_j^a \to D_j^t$. Then*

$$A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2 \stackrel{\text{def}}{=} \|_{i=1}^{n} \|_{j=1}^{m} (C_i^a \to C_i^t) \,\tilde{\oplus}_{\mathrm{A}}\, (D_j^a \to D_j^t), \tag{14}$$

where, if we define $C_{ij}^a \stackrel{\text{def}}{=} C_i^a \otimes D_j^a$ and $C_{ij}^t \stackrel{\text{def}}{=} C_i^t \oplus D_j^t$, we have

$$(C_i^a \to C_i^t) \,\tilde{\oplus}_{\mathrm{A}}\, (D_j^a \to D_j^t) \stackrel{\text{def}}{=} \begin{cases} \mathbf{1}_{\mathrm{A}}, & \text{if } C_{ij}^a \vdash C_{ij}^t; \\ C_{ij}^a \to (C_{ij}^a \otimes C_{ij}^t), & \text{otherwise.} \end{cases} \tag{15}$$

It is easy to see that this syntactic operation corresponds, at the semantic level, to an upper bound.

Proposition 38 *If A_1 and A_2 are as stated in* Definition 37*, then $A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2$ is in syntactic normal form. Further, we have both $[\![A_1]\!] \subseteq [\![A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2]\!]$ and $[\![A_2]\!] \subseteq [\![A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2]\!]$, that is, $A_1 \vdash_{\mathrm{A}} A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2$ and $A_2 \vdash_{\mathrm{A}} A_1 \,\tilde{\oplus}_{\mathrm{A}}\, A_2$.*

We now have an obvious merge operator which is completely determined by the underlying, base-level constraint system.

Definition 39 (Canonical ask-tell merge operator.) *Let* $\mathcal{A} \stackrel{\text{def}}{=} |\mathcal{A}(\bar{\mathcal{D}})|$. *The operator* $\hat{\oplus}_{\text{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ *given, for each* $A_1, A_2 \in \mathcal{A}$, *by*

$$A_1 \,\hat{\oplus}_{\text{A}}\, A_2 \stackrel{\text{def}}{=} \rho_{\text{A}}(A_1 \,\tilde{\oplus}_{\text{A}}\, A_2)$$

is called the canonical merge operator over $\mathcal{A}$ induced by $\bar{\mathcal{D}}$.

Unfortunately, the canonical merge operator turns out to be inaccurate, due to the normal form employed for agents. Consider the ask-and-tell construction applied to the *Con* domain of Section 5.1, and the agents in normal form [5]

$$A_1 \stackrel{\text{def}}{=} \mathbf{1} \to Z \parallel XZ \to XYZ \quad \text{and} \quad A_2 \stackrel{\text{def}}{=} X \to XY.$$

It easy to see that the canonical merge operator gives

$$A_1 \,\hat{\oplus}_{\text{A}}\, A_2 \stackrel{\text{def}}{=} \rho_{\text{A}}(A_1 \,\tilde{\oplus}_{\text{A}}\, A_2) = XZ \to XYZ.$$

If we consider the non-normalized agent $A_1' = \mathbf{1} \to Z \parallel X \to XYZ$, we have $[\![A_1']\!] = [\![A_1]\!]$ but $\rho_{\text{A}}(A_1' \,\tilde{\oplus}_{\text{A}}\, A_2) = X \to XY$, which is strictly stronger than $A_1 \,\hat{\oplus}_{\text{A}}\, A_2$. The problem can be tracked down, as in the case of the projection operators, to the introduction, by means of the absorption rule, of "unnecessary dependencies" needed to satisfy condition N_3 of the semantic normal form. However, while for projection operators we had a standard solution, here the situation is more difficult. As the example suggests, in order to define a precise merge operator we need mechanisms for

(i) *weakening* constraints (now we only *strengthen* them), and
(ii) *splitting* ask-tell pairs (now we only *combine* them).

We now give a general way of defining merge and widening operators for the ask-and-tell construction which are more precise than the canonical merge operator. First of all, let us deal with the problem of constraints' weakening.

Definition 40 (Weakening.) *An operation* $\circledcirc : \mathcal{D} \to \mathcal{D}$ *over a constraint system* $\bar{\mathcal{D}} = \langle \mathcal{D}, \otimes, \oplus, \mathbf{0}, \mathbf{1}, \{\bar{\exists}_{\bar{X}}\}, \{d_{\bar{X}\bar{Y}}\}\rangle$ *is said* a weakening operator for $\bar{\mathcal{D}}$ *if it satisfies, for each* $C, C_1, C_2 \in \mathcal{D}$*:*

Q_1. $(C_1 \circledcirc C_2) \otimes C_1 = C_1$,
Q_2. $(C_1 \circledcirc C_2) \otimes C_2 = C_1 \otimes C_2$,
Q_3. $(C_1 \circledcirc C_2) \circledcirc C_2 = C_1 \circledcirc C_2$.

[5] For simplicity we use juxtaposition instead of the usual set notation.

The intuitive explanation of this axiomatization is as follows. Condition Q_1, which can be restated as $C_1 \vdash C_1 \circledcirc C_2$, means that the weakening operation is correct (it does not add anything). Condition Q_2 states that weakenings are not too aggressive: a weakened constraint can be restored by *adding* what was *taken out*. Q_3 says that taking out twice the same thing is pointless. Observe that these conditions are very weak, while being sufficient for what follows.

Example 41 (Weakening over intervals.) *Consider a domain for numerical bounds analysis based on intervals. For instance, take the simple constraint system of* Section 4.2, *restricted to the intervals component, and apply to it the determinate c.s. construction. A weakening operator can be defined along the following lines, considering, for simplicity, only closed intervals:*

$$[\, l_1, u_1 \,] \circledcirc [\, l_2, u_2 \,] \stackrel{\text{def}}{=} [\, l, u \,],$$

where

$$l \stackrel{\text{def}}{=} \begin{cases} -\infty, & \text{if } l_1 \leq l_2; \\ l_1, & \text{otherwise;} \end{cases} \qquad \text{and} \qquad u \stackrel{\text{def}}{=} \begin{cases} +\infty, & \text{if } u_1 \geq u_2; \\ u_1, & \text{otherwise.} \end{cases}$$

Such an operator is easily verified being a weakening.

We are now in position to define a class of procedures for weakening the ask constraints of finite cc agents while preserving the semantics. As this operation is somewhat opposite to the absorption rewrite rule of our rewriting system, we call it *de-absorption*. This involves the *splitting* of ask-tell pairs.

Definition 42 (De-absorption step.) *Let $A = \|_{i=1}^{n} C_i^a \to C_i^t$ an agent in syntactic normal form over a c.s. $\bar{\mathcal{D}}$, and let $\circledcirc$ be a weakening over $\bar{\mathcal{D}}$. Then A' is obtained from A by means of a* de-absorption step based on $\circledcirc$ *if and only if $h, k \in \{1, \ldots, n\}$ are such that $C_h^a \vdash C_k^a$, the condition $C_h^a \neq (C_h^a \circledcirc C_k^t) \otimes C_k^a$ holds, and*

$$A' = \left((C_h^a \circledcirc C_k^t) \otimes C_k^a \to C_h^t \right) \| A.$$

Observe that any de-absorption step results in the strict weakening of an ask constraint. In fact, we have $C_h^a \vdash (C_h^a \circledcirc C_k^t)$ by Q_1, and $C_h^a \vdash C_k^a$ by hypothesis, thus $C_h^a \Vdash (C_h^a \circledcirc C_k^t) \otimes C_k^a$.

Definition 43 (De-absorption procedure.) *A* de-absorption procedure *is any algorithm transforming a finite* cc *agents in syntactic normal form which can be characterized as follows:*

phase 1. *transform the input agent A into A' by performing any number of de-absorption steps;*

phase 2. *transform A' into the output agent A'' by applying the rewriting system of* Figure 1 restricted to the structural and reduction rules.

A de-absorption procedure will be called maximal *if it applies all the possible de-absorption steps.*

It is now possible to prove the following result.

Theorem 44 *Any de-absorption procedure is semantics-preserving.*

In all those cases where we have a de-absorption procedure which is a function over $|\mathcal{A}(\bar{\mathcal{D}})|$ we have an obvious way to define a merge operator: by applying the *syntactic* merge operator of Definition 39 to de-absorbed agents.

Definition 45 (Merge operator with de-absorption.) *Let $\bar{\mathcal{D}}$ be a constraint system, and let $\delta_{\circledcirc} : |\mathcal{A}(\bar{\mathcal{D}})| \to \mathcal{A}(\bar{\mathcal{D}})$ be a de-absorption procedure. The* merge operator based on $\delta_{\circledcirc}$ *is given, for each $A_1, A_2 \in |\mathcal{A}(\bar{\mathcal{D}})|$, by*

$$A_1 \mathbin{\dot{\oplus}_{\mathrm{A}}} A_2 \stackrel{\text{def}}{=} \rho_{\mathrm{A}}\Big(\delta_{\circledcirc}(A_1) \mathbin{\tilde{\oplus}_{\mathrm{A}}} \delta_{\circledcirc}(A_2)\Big).$$

Any such operator, by virtue of Theorem 44 and Proposition 38, is clearly a merge operator in the sense of Definition 6. De-absorption procedures which are not functions are still useful for designing widening operators.

We now quickly show some examples of ask-and-tell constraint systems. For the more interesting things we have to wait until the next section, where combination of constraint domains are introduced.

7.2 More bounds and relations analysis for numeric domains

Ask-and-tell constraint systems are suitable for modeling approximate inference techniques which are very useful in a practical setting. Following Section 4.2, there is another technique which is used for the analysis described in [5,6,3]: *relational arithmetic* [38]. This technique allows to infer constraints on the qualitative relationship of an expression to its arguments. Consider the simple constraint system of Section 4.2, and apply to it the determinate construction of Section 5. Now apply the ask-and-tell construction to the result. Relational arithmetic can be described by a number of (concurrent) agents. Here are some of them, where x and y are arithmetic expressions, and $\bowtie$ ranges in $\mathsf{R} \stackrel{\text{def}}{=} \{=, \neq, \leq <, \geq, >\}$:

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

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.

7.3 Definiteness analysis: Def, Pos, and more

The prototypical example of data-flow analysis taking advantage of dependency information is definiteness analysis. In our setting a domain for definiteness can be obtained as follows. Take the *atomic* s.c.s. of Section 4.1. Apply to it the determinate construction as outlined in Section 5.1. Now apply the ask-and-tell construction to the result, with the merge operator obtained along the lines of Definition 45 choosing:

(i) diagonal elements like $d^{\mathrm{A}}_{XY} \stackrel{\mathrm{def}}{=} \{X\} \rightarrow \{X, Y\} \parallel \{Y\} \rightarrow \{X, Y\}$;
(ii) set-theoretic difference as weakening operator;
(iii) the maximal de-absorption procedure (i.e. the one which applies all the possible de-absorption steps).

It can be shown that the domain so obtained is *Def* [17,2]. Its elements can keep track of non-trivial dependencies like the ones induced by symbolic and numeric constraints. For example, the dependencies of $A = f(B, C)$ are captured by the agent $\{A\} \rightarrow \{B, C\} \parallel \{B, C\} \rightarrow \{A\}$. This example gives us the possibility of pointing out that the entire business of weakenings and de-absorption procedures is not something we can easily avoid. When using definite sentences to represent dependencies, as in our case and in the representations for *Def* studied in [2], obtaining a maximal weakening of the antecedents is crucial for obtaining precise merge operators, let alone for computing the join when it exists. Our present requirement of employing maximal de-absorption corresponds to the requirement, in the representations studied in [2], of the sentences being in *orthogonal* form (which has its costs, since orthogonality must be obtained and preserved by all the domain's operations). In [2] it is also presented a merge operator, for the representation RCNF_{Def}, intended to trade precision for efficiency. It does that by not insisting on orthogonality, which in our setting corresponds to the use of a partial de-absorption procedure.

Pos is (like *Def*) a domain of boolean functions [10,2]. It consists precisely of those functions assuming the true value under the *everything-is-true* assignment. In [2] it is shown that *Pos* is strictly more precise than *Def* for

groundness analysis. If we apply the powerset construction of Section 6 to the ask-and-tell c.s. of the previous section we obtain a very precise (and complex) domain for simple dependencies. In [22] it is referred to as $\mathcal{U}(Def)$ (where $\mathcal{U}$ denotes disjunctive completion) and is shown to be equivalent to $\mathcal{U}(Pos)$. On the other hand, in [20] it has been shown that $\mathcal{U}(Pos)$ is strictly more precise than Pos, even though this extra-precision is not needed for definiteness analysis.

8 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 [13,14]. 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, or, in other words, it is not synchronized with the domain's operations. This is achieved by considering ask-and-tell constraint systems built over *product* 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 [11], which is directly based onto low-level actions such as *tests* and *queries*. While the approach in [11] is immediately applicable to a wider range of analyses (including the ones dealing with non-monotonic properties) the approach we follow here for our restricted set of analyses has the merit of being much more elegant. We start with a finite set of constraint systems 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 given constraint systems.

A product constraint system can easily be obtained: given the constraint systems $\bar{\mathcal{D}}_1, \ldots, \bar{\mathcal{D}}_n$ just consider their algebraic direct product (where all the operations and relations are defined point-wise). An alternative way of obtaining a product constraint system is to start from a collection of simple constraint systems and then applying the determinate construction.

Definition 46 (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*

$$\prod_{i=1}^{n} \mathcal{S}_i \stackrel{\text{def}}{=} \langle \mathcal{C}_\times, \vdash_\times, \bot_\times, \top_\times \rangle,$$

where the product tokens are

$$\begin{aligned}
\mathcal{C}_\times \stackrel{\text{def}}{=} & \left\{ (c_1, \top_2, \ldots, \top_n) \;\middle|\; c_1 \in \mathcal{C}_1 \right\} \\
& \cup \left\{ (\top_1, c_2, \top_3, \ldots, \top_n) \;\middle|\; c_2 \in \mathcal{C}_2 \right\} \\
& \vdots \\
& \cup \left\{ (\top_1, \ldots, \top_{n-1}, c_n) \;\middle|\; c_n \in \mathcal{C}_n \right\} \\
& \cup \{\bot_\times\},
\end{aligned}$$

$\bot_\times \stackrel{\text{def}}{=} (\bot_1, \ldots, \bot_n)$, $\top_\times \stackrel{\text{def}}{=} (\top_1, \ldots, \top_n)$, *and the product entailment is defined as the least relation satisfying conditions* E_1–E_5 *of* Definition 19 *and the following ones, for each* $C \in \wp_f(\mathcal{C}_\times)$*:*

$$\begin{array}{ccc}
\pi_1(C) \vdash_1 c_1 & \Longrightarrow & C \vdash_\times (c_1, \top_2, \ldots, \top_n) \\
\vdots & \vdots & \vdots \\
\pi_n(C) \vdash_n c_n & \Longrightarrow & C \vdash_\times (\top_1, \ldots, \top_{n-1}, c_n).
\end{array}$$

Taking the product of constraint systems, we have realized the simplest form of domain combination. It corresponds to the direct product construction of [13], 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 of the component domains. 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 [27] non-linear constraints (like $X = Y * Z$) 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 bottom-up 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 $ground^\natural(X)$ 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

$$\mathrm{ask}\Big(ground^\natural(Y); ground^\natural(Z)\Big) \rightarrow \mathrm{tell}(X = Y * Z).$$

In fact, any CLP($\mathcal{R}$) user *must* know that $X = Y * Z$ is just a shorthand for that agent! (A similar treatment can 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 4.2 and 7.2, when the abstract constraint system extracts information from non-linear constraints, for example with the agent

$$A = \mathrm{ask}(Y > 0 \wedge Z > 0 \wedge Y \bowtie 1) \rightarrow \mathrm{tell}\Big((Y * Z) \bowtie Z\Big)$$

of 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 the numeric abstract constraint system with one for definiteness (by the product and the ask-and-tell constructions) and using, instead of A, the agent

$$A' = \mathrm{ask}\Big(ground^\sharp(Y); ground^\sharp(Z)\Big) \rightarrow A.$$

Beware not to confuse $ground^\natural(X)$ with $ground^\sharp(X)$. The first is the *concrete one*: X is definite if and only if $ground^\natural(X)$ is entailed in the current concrete store. In contrast, having $ground^\sharp(X)$ entailed in the *abstract* constraint store at some program point, and assuming a correct definiteness analysis, means that X is certainly bound to a unique value in the concrete computation at that program point. The vice-versa, of course, does not necessarily hold.

Let us see another example. The analysis described in [24] 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) [25]. 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 [24] 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. bounds analysis), something which is not captured by common definiteness analyses while being crucial when dealing with non-linear constraints. If we take the ask-and-tell construction over the product of a constraint system for definiteness with a numerical one we can solve the problem. $X = Y * Z$ would be *abstractly compiled* into the agent

$$\begin{aligned}
&\mathrm{ask}\Big(ground^\sharp(Y) \wedge ground^\sharp(Z)\Big) \rightarrow \mathrm{tell}\Big(ground^\sharp(X)\Big) \\
\| &\,\mathrm{ask}(Y = 0; Z = 0) \rightarrow \mathrm{tell}\Big(ground^\sharp(X)\Big) \\
\| &\,\mathrm{ask}\Big(ground^\sharp(X) \wedge ground^\sharp(Z) \wedge Z \neq 0\Big) \rightarrow \mathrm{tell}\Big(ground^\sharp(Y)\Big) \\
\| &\,\mathrm{ask}\Big(ground^\sharp(X) \wedge ground^\sharp(Y) \wedge Y \neq 0\Big) \rightarrow \mathrm{tell}\Big(ground^\sharp(Z)\Big).
\end{aligned}$$

Of course, this is much more precise than the *Def* formula $X \leftarrow Y \wedge Z$. Observe that, when analyzing CLP($\mathcal{R}$) programs, there is a bidirectional flow of information: definiteness information is required for a correct handling of delayed constraints and thus for deriving more precise numerical patterns which, in turn, are used to provide more precise definiteness information. There is another obvious way in which numerical bounds and relations improve definiteness (and any other analysis, indeed): by excluding computation paths which are doomed to fail (this is modeled in a domain-independent way by the *smash agent* seen above). We are thus requiring a quite complicated interaction between domains. It is even more complicated if you consider that the numerical component we have sketched is the combination (in the sense of the present section) of a domain for intervals with one for arithmetic relationships (even though, for simplicity, it was not presented in that way).

The techniques we propose are suitable for approximating the behavior of several common built-ins. Consider, for instance, the `functor/3` built-in. Consider a product constraint system with four components: one for *simple types* (where tokens like $compound(X)$ or $atom(X)$ indicate that the variable X is bound to take Herbrand compounds or constants, respectively) one for definiteness, one incorporating numerical information (including at least signs, e.g. tokens of the kind $X \geq 0$, $X > 0$ and $X = 0$), and one involving symbolic, structural information. Then, the (success) semantics of `functor(T, F, N)` can be ap-

proximated easily and quite precisely by means of the following finite agent over the product:

$$\begin{aligned}
&\text{tell}\Big(term(T), atom(F), ground(F), integer(N), N \geq 0, ground(N)\Big) \parallel \\
&\text{ask}\Big(atom(T); N = 0; T = F\Big) \rightarrow \\
&\qquad \text{tell}\Big(atom(T), ground(T), N = 0, T = F\Big) \parallel \\
&\text{ask}\Big(compound(T); N > 0; T \neq F\Big) \rightarrow \\
&\qquad \text{tell}\Big(compound(T), N > 0, T \neq F\Big).
\end{aligned}$$

9 Conclusion and future work

We have shown a notion of constraint system which is general enough to encompass both the concrete domains of computation of actual constraint logic-based languages, and several of their abstract interpretations useful for data-flow analysis. We have also shown how these constraint systems are integrated within an appropriate framework for the definition of non-standard semantics of constraint logic-based languages. Some significant members of the introduced class of constraint systems have been presented, together with construction techniques which induce a hierarchy of domains. These domains have several nice features from a theoretical point of view. In particular, we have proposed a general methodology for domain combination with asynchronous interaction. In this kind of combinations the communication among domains can be expressed in a very simple way. The methodology also inherits all the semantic elegance of concurrent constraint programming languages, which provide the basis on which it is built. Future work includes studying in depth the problem of the semantic normal form for finite cc agents, both in general and in particular cases. The aim is to find more satisfactory solutions to the problem of merging finite cc agents. We also would like to answer the following question: are there variations of these ideas which are applicable also to analysis oriented towards "non-logical" properties?

Acknowledgement

I wish to thank Maurizio Gabbrielli, Roberto Giacobazzi, Giorgio Levi, Catuscia Palamidessi, Francesca Rossi, Vijay Saraswat, and Enea Zaffanella for the discussions we had on the subject and for reading draft versions of this paper. Enea was especially kind in helping me with some technical issues. Special thanks to Giorgio for his extensive support.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] T. Armstrong, K. Marriott, P. Schachte, and H. Sondergaard. Two classes of boolean functions for dependency analysis. Technical Report 94/211, Dept. Computer Science, Monash University, Melbourne, 1994.

[3] R. Bagnara. On the detection of implicit and redundant numeric constraints in CLP programs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the "1994 Joint Conference on Declarative Programming (GULP-PRODE '94)"*, pages 312–326, Peñíscola, Spain, September 1994.

[4] R. Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. Technical Report TR-96-10, Dipartimento di Informatica, Università di Pisa, 1996.

[5] R. Bagnara, R. Giacobazzi, and G. Levi. Static analysis of CLP programs over numeric domains. In M. Billaud, P. Castéran, MM. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes "Workshop on Static Analysis '92"*, volume 81–82 of *Bigre*, pages 43–50, Bordeaux, September 1992. Extended abstract.

[6] R. Bagnara, R. Giacobazzi, and G. Levi. An application of constraint propagation to data-flow analysis. In *Proceedings of "The Ninth Conference on Artificial Intelligence for Applications"*, pages 270–276, Orlando, Florida, March 1993. IEEE Computer Society Press, Los Alamitos, CA.

[7] R. Barbuti and A. Martelli. A structured approach to semantics correctness. *Science of Computer Programming*, 3:279–311, 1983.

[8] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 114–129, Vancouver, Canada, 1993. The MIT Press.

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

[10] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[11] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, January 1994.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[14] P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[15] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992.

[16] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, Berlin, 1992.

[17] P. Dart. On derived dependencies and connected databases. *Journal of Logic Programming*, 11(2):163–188, 1991.

[18] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[19] S. K. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *The Journal of Logic Programming*, 18(2):149–176, February 1994.

[20] G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium (ILPS '94)*, pages 655–669. The MIT Press, 1994.

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

[22] R. Giacobazzi and F. Ranzato. Compositional optimization of disjunctive abstract interpretations. In H.R. Nielson, editor, *Proc. of the 1996 European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 141–155. Springer-Verlag, Berlin, 1996.

[23] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.

[24] M. Hanus. Analysis of nonlinear constraints in CLP($\mathcal{R}$). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Progra mming*, pages 83–99, Budapest, Hungary, 1993. The MIT Press.

[25] 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, Cambridge, Mass., 1993.

[26] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[27] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[28] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994.

[29] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.

[30] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[31] C. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2:43–66, 1985.

[32] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.

[33] V. A. Saraswat. The category of constraint systems is cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, 1992.

[34] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press Cambridge, Mass., 1993.

[35] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 333–353. ACM, 1991.

[36] D. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Proc. ninth Int. Coll. on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, Berlin, 1982.

[37] M. C. Scudellari. Analisi bottom-up di programmi logici con vincoli basata su trasformazioni *magic-set*. M.Sc. dissertation, R. Bagnara and G. Levi supervisors, University of Pisa, April 1996. In Italian, available on WWW at URL `http://www.di.unipi.it/~bagnara`.

[38] R. Simmons. Commonsense arithmetic reasoning. In *Proc. AAAI-86*, pages 118–124, 1986.

[39] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint logic programming over finite domains: the design, implementation, and applications of `cc(fd)`. Technical report, Brown University, Providence, RI, 1992.