Domain Independent Ask Approximation in CCP*

Enea Zaffanella

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
zaffanel@di.unipi.it

Abstract. The main difficulty in the formalization of a static analysis framework for CC programs is probably related to the correct approximation of the entailment relation between constraints. This approximation is needed for the abstract evaluation of the ask guards and directly influences the overall precision of the analysis. In this paper we provide a solution to this problem by stating reasonable correctness conditions relating the abstract and the concrete domains of computation. The solution is domain independent in the sense that it can be applied to the class of *downward closed* observables. Properties falling in this class have already been studied in the context of the analysis of sequential (constraint) logic programs. As an example, we consider an abstract domain designed for the analysis of *freeness* in CLP programs and we show how it can be usefully applied in the CC context to discover undesired data dependencies between concurrent processes.

1 Introduction

Abstract interpretation is intended to formalize the idea of approximating program properties by evaluating them on suitable non-standard domains. The standard domain of values is replaced by a domain of descriptions of values and the basic operators are provided with a corresponding non-standard interpretation. In the classical framework of abstract interpretation [7], the relation between abstract and concrete semantic objects is provided by a pair of adjoint functions referred to as *abstraction* α and *concretization* γ. The idea is to describe data-flow information about a program P by evaluating the program by means of an abstract interpreter $\mathcal{I}$. The abstract interpretation $\mathcal{I}(P)$ is *correct* if any possible concrete computation is described by $\gamma(\mathcal{I}(P))$.

Concurrent Constraint (CC) programming [26] arises as a generalization of both concurrent logic programming and constraint logic programming (CLP). In the CC framework processes are executed concurrently in a shared *store*, which is a constraint representing the global state of the computation. Communication

* This work has been supported by the "PARFORCE" (Parallel Formal Computing Environment) BRA-Esprit II Project n. 6707.

is achieved by ask and tell basic actions. A process telling a constraint simply adds it to the current store, in a completely asynchronous way. Synchronization is achieved through *blocking asks*. Namely the process is suspended when the store does not entail the ask constraint and it remains suspended until the store entails it. While being elegant from a theoretical point of view, this synchronization mechanism turns out to be very difficult to model in the context of static analysis. The reason for such a problem lies in the anti–monotonic nature of the ask operator wrt the asked constraint: if we replace this constraint with a weaker one we obtain stronger observables. As a consequence, the approximation theory developed to correctly characterize the *upward closed* properties (i.e. properties closed wrt entailment) becomes useless when we are looking for a domain independent solution to the ask approximation problem [28].

In this paper we thus consider the *downward closed* properties and we specify suitable domain independent correctness conditions that allow to overcome the problem of a safe abstraction of ask constraints. In particular we develop an approximation theory that correctly detects the definite suspension of an ask guard. This information can be used in many ways, e.g. for the debugging of `CC` programs as well as to identify processes that are definitely serialized (so that we can avoid their harmful parallel execution). Moreover, the same information can improve the precision of the static analysis framework, as it allows to cut the branches of code that will not be considered in the concrete computation.

This (partial) classification of `CC` program's observables is not new. See [19] for an interesting discussion about *safety* and *liveness* properties in `CCP`, being downward closed and upward closed respectively. As a matter of fact, in the literature there already exist abstract domains developed for the static analysis of sequential (constraint) logic languages dealing with downward closed observables, e.g. freeness in the Herbrand constraint system [23, 6, 3] as well as in arithmetic constraint systems [13, 20]. It is our opinion that these abstract domains can be usefully applied to the `CC` context and provide meaningful ask approximations. We indeed show an example where the abstract domain formalized in [13] is applied to detect an *undesired* data dependency between two concurrent processes.

2 Preliminaries

Throughout the paper we will assume familiarity with the basic notions of lattice theory [2] and abstract interpretation [7, 9].

A set P equipped with a partial order $\leq$ is said to be *partially ordered*. Given a partially ordered set $\langle P, \leq \rangle$ and $X \subseteq P$, the set $\uparrow X = \{y \in P \mid \exists x \in X \,.\, x \leq y\}$ is the *upward closure* of X. In particular X is an upward closed set iff $X = \uparrow X$. The *downward closure* $\downarrow X$ and downward closed sets are defined dually.

We write $f : A \to B$ to mean that f is a total function of A into B. Functions from a set to the same set are usually called *operators*. The identity operator $\lambda x.x$ is denoted by id. Given the partially ordered sets $\langle A, \leq_A \rangle$ and $\langle B, \leq_B \rangle$, a function $f : A \to B$ is *monotonic* if for all $x, x' \in A.\, x \leq_A x'$ implies $f(x) \leq_B f(x')$.

f is *continuous* iff for each non-empty chain $X \subseteq A$: $f(\sqcup_A X) = \sqcup_B f(X)$. A function f is *additive* iff the previous conditions are satisfied for each non-empty set $X \subseteq A$ (f is also called *complete join-morphism*). A *retraction* ϱ on a partially ordered set $\langle L, \leq \rangle$ is a monotonic operator such that for all $x \in L$. $f(f(x)) = f(x)$ (idempotent). An *upper closure operator* (uco) on L is a retraction ρ such that $\forall x \in L.\, x \leq \rho(x)$ (extensive); a *lower closure operator* (lco) on L is a retraction δ such that $\forall x \in L.\, \delta(x) \leq x$ (reductive). More on closure operators can be found in [8].

Let $\langle L, \leq, \bot, \top, \vee, \wedge \rangle$ and $\langle L', \leq', \bot', \top', \vee', \wedge' \rangle$ be complete lattices. An *upper Galois connection* between L and L' is a pair of functions (α, γ) such that

1. $\alpha : L \to L'$ and $\gamma : L' \to L$
2. $\forall x \in L\,. \forall y \in L'\,. \alpha(x) \leq' y \Leftrightarrow x \leq \gamma(y)$.

An upper Galois *insertion* between L and L' is an upper Galois connection such that α is surjective (equivalently, γ is one-to-one). Both α (the abstraction function) and γ (the concretization function) are monotonic. α is a *complete join-morphism* and γ is a *complete meet-morphism* and each one determines the other; i.e. $\alpha(x) = \wedge' \{ y \in L' \mid x \leq \gamma(y) \}$ and $\gamma(y) = \vee \{ x \in L \mid \alpha(x) \leq' y \}$.

3 The language

cc is not a language, it is a class of languages parametric wrt the constraint system, a semantic domain formalizing the gathering and the management of partial information. Starting from Scott's partial information systems [27], describing the basic notion of entailment in a constructive fashion, the domains of [26] enclose typical cylindric algebras' operators [17].

Definition 1 (partial information system).
A *partial information system* is a quadruple $\langle D, \Delta, Con, \vdash \rangle$ where D is a denumerable set of elementary assertions (tokens), $\Delta \in D$ is a distinguished assertion (the least informative token), Con is a family of *finite* subsets of D (the consistent subsets of tokens) and $\vdash \subseteq Con \times Con$ is a (compact) entailment relation satisfying (for $u, v, w \in Con$, $P \in D$):

$$\begin{array}{l} \emptyset \vdash \{\Delta\} \\ u \vdash \{P\} \text{ if } P \in u \\ u \vdash w \quad \text{if } u \vdash v \text{ and } v \vdash w \end{array}$$

Entailment closed sets of tokens are called *constraints* and provide representatives for the equivalence classes induced by the entailment relation; in particular, *true* denotes the set of the trivial tokens. The *simple constraint system* generated by the partial information system is the set of all the constraints together with the partial order induced on them by the reverse of the entailment relation (which we will denote $\dashv$). We write $\otimes$ to denote the constraint composition operator (the *lub*) which is obtained by taking the entailment closure of the set theoretical union. We refer to [27] and [26] for a more detailed presentation.

Progr	::=	Dec . Agent
Dec	::=	ϵ
	|	p(x):-Agent. Dec
Agent	::=	Stop
	|	tell(c)
	|	$\exists$ x in Agent
	|	Agent $\|\|$ Agent
	|	$\sum_{i=1}^{n}$ ask(c_i)->Agent$_i$
	|	p(y)

Table 1. The syntax

Definition 2 (constraint system).
A (cylindric) *constraint system* $\mathcal{C}^{\top} = \langle\, C \cup \{false\}, \dashv, true, false, \otimes, \sqcap, V, \exists_x, d_{xy} \,\rangle$ is an algebraic structure where

- $\langle\, C, \dashv, true, \otimes, \sqcap \,\rangle$ is a simple constraint system
- $false$ is the top element
- V is a denumerable set of variables
- $\forall x, y \in V$, $\forall c, d \in C$, the cylindric operator $\exists_x$ satisfies
  1. $\exists_x false = false$
  2. $\exists_x c \dashv c$
  3. $c \dashv d$ implies $\exists_x c \dashv \exists_x d$
  4. $\exists_x (c \otimes \exists_x d) = \exists_x c \otimes \exists_x d$
  5. $\exists_x (\exists_y c) = \exists_y (\exists_x c)$
- $\forall x, y, z \in V$, $\forall c \in C$, the diagonal element d_{xy} satisfies
  1. $d_{xx} = true$
  2. $z \not\equiv x, y$ implies $d_{xy} = \exists_z (d_{xz} \otimes d_{zy})$
  3. $x \not\equiv y$ implies $c \dashv d_{xy} \otimes \exists_x (c \otimes d_{xy})$

Note that we are distinguishing between the consistent constraints C and the top element $false$ representing inconsistency. In the following we will write $\mathcal{C}$ to denote the subalgebra of consistent constraints, namely the set C together with the constraint system's operators restricted to work on C. We will denote operators and their restrictions in the same way and we will often refer to $\mathcal{C}$ as a "constraint system".

Tables 1 and 2 introduce the syntax and the operational semantics of `CC` languages. For notational convenience, we consider processes having one variable only in the head. We also assume that for all the procedure names occurring in the program text there is a corresponding definition. The operational model is

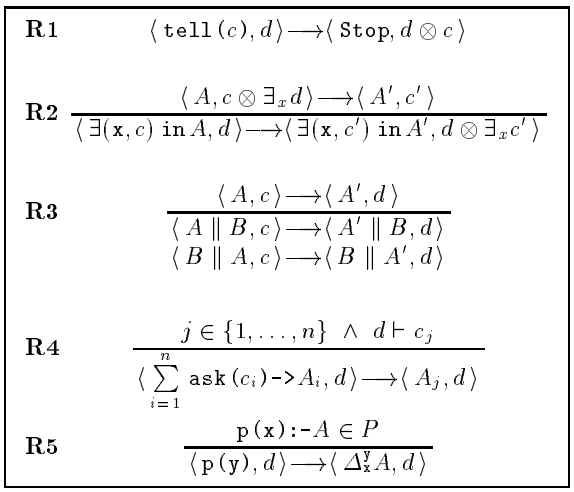

Table 2. The transition system T

described by a transition system $T = (Conf, \longrightarrow)$. Elements of $Conf$ (configurations) consist of an agent and a constraint, representing the residual computation and the global store respectively. $\longrightarrow$ is the (minimal) transition relation satisfying axioms **R1-R5**.

The execution of an elementary tell action simply adds the constraint c to the current store d (no consistency check). Axiom **R2** describes the hiding operator. The syntax is extended to deal with a local store c holding information about the hidden variable x. Hence the information about x produced by the external environment does not affect the process behaviour and conversely the external environment cannot access the local store. Initially the local store is empty, i.e. $\exists\, \mathtt{x}\ \mathtt{in}\ A \equiv \exists(\mathtt{x}, true)\ \mathtt{in}\ A$. Parallelism is modelled as *interleaving* of basic actions. In a guarded choice operator, a branch A_i is enabled in the current store d iff the corresponding guard constraint $\mathrm{ask}(c_i)$ is entailed by the store, i.e. $d \vdash c_i$. The guarded choice operator indeterministically selects one enabled branch A_i and behaves like it. If there is no enabled branch then it suspends, waiting for other processes to add the desired information to the store. Finally, when executing a procedure call, rule **R5** models parameter passing without variable renaming, where $\mathtt{p(x)}\mathtt{:-}A \in P$ and $\Delta^{\mathtt{y}}_{\mathtt{x}}\, A = \exists\, \mu\ \mathtt{in}\, (\mathtt{tell}(d_{\mu y}) \parallel \exists\, \mathtt{x}\ \mathtt{in}\, (\mathtt{tell}(d_{x\mu}) \parallel A))$[2].

Definition 3 (c-computations semantics).
A c-computation for program $D.A$ is a sequence $s = \langle A_0, c_0\rangle \ldots \langle A_i, c_i\rangle \ldots$ of configurations such that $A_0 = A$ and $c_0 = c$ and for all $0 < i < |s|$

[2] Here μ is a variable not occurring in the program [26].

$\langle A_{i-1}, c_{i-1}\rangle \longrightarrow \langle A_i, c_i\rangle$[3]. The c-computations semantics of a program is the set of all its c-computations.

Let $\not\longrightarrow$ denote the absence of admissible transitions. Computations reaching configuration $\langle A_n, c_n\rangle$ such that $\langle A_n, c_n\rangle \not\longrightarrow$ are called *finite* computations. If the residual agent A_n contains some choice operators then the corresponding computation is *suspended*, otherwise it is a *successful* computation and in this case we denote A_n by ϵ.

Definition 4 (c.a.c. semantics).
The c.a.c. (computed answer constraints) semantics for program $P = D.A$ in the store c is

$$\mathcal{O}[\![D.A]\!]c = \left\{ d \in C \;\middle|\; \langle A\ ,\ c\rangle \xrightarrow{*} \langle B\ ,\ d\rangle \not\longrightarrow \right\} \cup \left\{ d \in C \;\middle|\; \begin{array}{l} A_0 = A, \qquad c_0 = c, \qquad d = \bigotimes_{i<\omega} c_i, \\ \langle A_0\ ,\ c_0\rangle \longrightarrow \ldots \longrightarrow \langle A_i\ ,\ c_i\rangle \longrightarrow \ldots \end{array} \right\}$$

Note that this semantics collects the limit constraints of infinite fair computations as well as the answer constraints associated to finite computations, regardless of whether the latter are successful or suspended. In any case we are considering consistent constraints only, i.e. we disregard all computations delivering *false*.

4 Program properties and approximations

As we have seen, the c.a.c. semantics of a CC program associates each initial store c to the set of all the consistent constraints that we obtain by executing $P = D.A$ at c. In a similar way we define a *semantic property* ϕ as a subset of C, namely the set of consistent constraints that satisfy the property. Therefore a program satisfies a semantic property ϕ at c iff the observables of the program are a *subset* of the property, i.e. $\mathcal{O}[\![P]\!]c \subseteq \phi$. Following this general view[4], the static analysis of a CC program can be formalized as a finite construction of an approximation (a superset) of the program denotation. If the approximation satisfies the semantic property, then we can correctly say that our program satisfies the property too. Abstract interpretation [9] formalizes the approximation construction process by mapping concrete semantic objects and operators into corresponding abstract semantic objects and operators.

Let us define a program property to be *ordering closed* iff it is downward closed or upward closed wrt entailment. As an example, consider the Herbrand constraint system $\mathcal{C}_H$. If the constraint $c \in C_H$ binds variable x to a ground term, then all the constraints $d \in C_H$ such that $d \vdash c$ will bind x to a ground term; therefore *groundness* is an upward closed property. On the other hand,

[3] As usual, if $|s| = \omega$ we also require that s is *fair* wrt the parallel operator.
[4] The same reasoning can be lifted in order to consider the c-computations semantics.

freeness is a downward closed property. A variable x is free in $c \in C_H$ iff there does not exist a term functor f/n such that $c \vdash (\exists_{y_1} \ldots \exists_{y_n} x = f(y_1, \ldots, y_n))$. Thus, if x is free in c then it will be free in all the constraints $d \in C_H$ such that $c \vdash d$. However, there obviously exist properties falling in none of these two classes, e.g. *independence*. Let us say that variables x and y share in $c \in C_H$ iff c binds x and y to the terms t_x and t_y such that $var(t_x) \cap var(t_y) \neq \emptyset$. Variables x and y are independent in c if they do not share in c. Now, if x and y share in c, we can choose constraints $d_1, d_2 \in C_H$ such that $d_1 \vdash c \vdash d_2$ and x and y are independent in both d_1 and d_2.

Ordering closed properties are very common in the static analysis of logic languages and furthermore they are easier to verify, because correctness of the abstract interpretation can be based on a semantics returning ordering closed observables. In [28] entailment closed[5] properties are considered. The main result is that it is impossible to develop a meaningful generalized semantics for `CC` languages in the style of [16], namely the only way to correctly abstract ask constraints in a domain independent fashion is a trivial approximation.

In this work we turn our interest upon downward closed properties and we show that a (carefully chosen but natural) notion of correctness of the abstract domain wrt the concrete one allows to automatically derive a correct approximation of all the asks occurring in the program. Dealing with such a class of properties, the collecting semantics can be defined naturally as the downward closure of the operational semantics, as there is no benefit in considering a stronger one [28].

Remark. If ϕ is downward closed then $\mathcal{O}[\![P]\!]c \subseteq \phi \;\Leftrightarrow \downarrow(\mathcal{O}[\![P]\!]c) \subseteq \phi$.

As we are observing infinite computations also, we have to be careful when defining the downward closed properties that we are interested in. In particular we have to remember that usually the correctness of our abstract semantic construction is based on the Scott's induction principle; this principle is only valid for *admissible* properties.

Definition 5. A property $\phi \subseteq C$ is *admissible* iff ϕ is closed under directed *lub*'s.

This definition means that whenever an admissible property is satisfied by all the finite approximations of the semantics, then *the* semantics will satisfy the property too. As an example of a downward closed property that is not admissible, consider the following definition of *nongroundness*: a variable x is nonground in $c \in C_H$ iff c binds x to a term t such that $var(t) \neq \emptyset$. Given the infinite chain of constraints $c_i \equiv (\exists_y x = f^i(y)) \in C_H$, for every $i < \omega$ we have that x is nonground in c_i. However, considering the limit constraint $c \equiv \bigotimes_{i<\omega} c_i = (x = f^\omega)$ one observes that x is *not* nonground in c. In order to grant the correctness of

[5] Due to a dual definition of the ordering on the constraint system, in [28] entailment closed properties are the downward closed ones. The choice of turning the domain upside–down was influenced by the standard theory of semantic approximation by means of upper Galois insertions [9].

this analysis, we have to redefine the property, e.g. by stating that if c binds x to an infinite term then x is nonground in c.

Hence, in this work we are interested in downward closed and admissible program properties. The Hoare's powerdomain construction [24, 27] over the constraint system characterizes this kind of observables.

Definition 6. The Hoare's powerdomain of the constraint system $\mathcal{C}$ is

$$\mathcal{H}(\mathcal{C}) = \langle\, \mathcal{P}{\downarrow}(C), \subseteq, \{true\}, C, \uplus, \cap \,\rangle$$

where $\mathcal{P}{\downarrow}(C)$ is the set of all the nonempty, downward closed and admissible subsets of C; $\uplus$ is the closure under directed C-lub's of the set theoretical union; $:\{\cdot\}: : C \to \mathcal{P}{\downarrow}(C)$ defined as $:\{c\}: = {\downarrow}\{c\}$ is the singleton embedding function.

The alert reader would observe that this collecting semantics models nonempty observables only. From a semantic construction point of view, this is not completely satisfactory as we cannot describe the behaviour of a program having inconsistent computations only. However, the alternative choice of considering failed computations would imply some negative consequences. Firstly, it would complicate the formalization of the correctness conditions, requiring a special treatment for inconsistency. Moreover it would degrade the precision of our static analysis, adding very little to the understanding of the program. To see this, observe that when considering downward closed observables a failed computation has to be interpreted as "the program *may* fail", meaning that anything can happen. Also consider that there are `cc` languages explicitly designed to statically avoid the possibility of a failing computation (see [25] for a discussion of this topic in the distributed programming context).
From now on, $\tilde{\otimes}$ and $\tilde{\exists}_x$ will denote the additive extensions of $\otimes$ and $\exists_x$ over $\mathcal{H}(\mathcal{C})$. Thus, for all $S, T \in \mathcal{P}{\downarrow}(C)$, we have

$$\begin{aligned} S \,\tilde{\otimes}\, T &= \biguplus \{\, :\{c \otimes d\}: \;\big|\; c \in S,\, d \in T,\, c \otimes d \in C \,\} \\ \tilde{\exists}_x\, S &= \biguplus \{\, :\{\exists_x c\}: \;\big|\; c \in S \,\} \end{aligned}$$

Note that the *merge over all paths* operator [9] is provided by $\uplus$ (the *lub* of $\mathcal{H}(\mathcal{C})$). Also note that in general the (lifted) constraint composition operator $\tilde{\otimes}$ is not idempotent, while being extensive.

5 Correctness

In this section we formalize the notion of *correctness* of an abstract domain wrt a concrete constraint system when downward closed properties are observed.

Definition 7. An abstract domain $\mathcal{A} = \langle\, L, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp, \otimes^\sharp, V, \exists^\sharp_x, d^\sharp_{xy} \,\rangle$ is a complete lattice $\mathcal{L} = \langle\, L, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \,\rangle$ together with a binary operator $\otimes^\sharp$, a family of unary operators $\exists^\sharp_x$ for $x \in V$ and a family of distinguished elements $d^\sharp_{xy} \in L$ for $x, y \in V$.

As outlined in the previous section, we have to grant the existence of an upper Galois insertion relating the Hoare's powerdomain of the concrete constraint system and the abstract domain of descriptions, together with suitable correctness conditions regarding the domain's operators.

Definition 8. An abstract domain $\mathcal{A} = \langle\, L, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp, \otimes^\sharp, V, \exists^\sharp_x, d^\sharp_{xy} \,\rangle$ is *down–correct* wrt the constraint system $\mathcal{C} = \langle\, C, \dashv, \mathit{true}, \otimes, \sqcap, V, \exists_x, d_{xy} \,\rangle$ using α iff there exists an upper Galois insertion (α, γ) relating $\mathcal{H}(\mathcal{C})$ and $\mathcal{L}$ and $\forall S, T \in \mathcal{P}{\downarrow}(C)$, $\forall x, y \in V$

$$\begin{array}{ll} \alpha(S \,\tilde{\otimes}\, T) & \sqsubseteq^\sharp\ \alpha(S) \otimes^\sharp \alpha(T) \\ \alpha(\tilde{\exists}_x S) & \sqsubseteq^\sharp\ \exists^\sharp_x \alpha(S) \\ \alpha(:\{d_{xy}\}:) & \sqsubseteq^\sharp\ d^\sharp_{xy} \end{array}$$

By assuming that the abstract domain $\mathcal{A}$ is *down–correct* wrt the constraint system $\mathcal{C}$ using α, we are able to prove the correctness of any abstract semantic construction based on the abstract interpretation theory. This means that the proof is valid for any abstract semantics that systematically mimics the basic concrete semantic operators ($\uplus$, $\otimes$, $\exists_x$, d_{xy}) and the relation $\dashv$ by using the corresponding abstract operators ($\sqcup^\sharp$, $\otimes^\sharp$, $\exists^\sharp_x$, $d^\sharp_{xy}$) and the relation $\sqsubseteq^\sharp$. For the purposes of the present work it is sufficient to consider the operational semantics.

Definition 9. Given the concrete agent (resp. program, configuration) A, the corresponding abstract agent (resp. program, configuration) $\alpha(A)$ is obtained by replacing all the concrete constraints $c \in C$ occurring in A by the corresponding abstractions $\alpha(:\{c\}:) \in L$. Abstract agents (resp. programs, configurations) are partially ordered by writing $A^\sharp \sqsubseteq^\sharp B^\sharp$ iff $B^\sharp$ is obtained from $A^\sharp$ by replacing each abstract constraint $c^\sharp$ by another abstract constraint $d^\sharp$ such that $c^\sharp \sqsubseteq^\sharp d^\sharp$.

The following lemma shows that the abstract program correctly mimics each transition of the concrete one. This also means that if the abstract program suspends, then the concrete program suspends too.

Lemma 10 (correctness).

$$\mathit{If}\ \begin{cases} \langle\, A, c \,\rangle \longrightarrow \langle\, B, d \,\rangle \\ \mathit{and} \\ \alpha(\langle\, A, c \,\rangle) \sqsubseteq^\sharp \langle\, A^\sharp, c^\sharp \,\rangle \end{cases} \quad \mathit{then} \quad \begin{cases} \langle\, A^\sharp, c^\sharp \,\rangle \longrightarrow \langle\, B^\sharp, d^\sharp \,\rangle \\ \mathit{and} \\ \alpha(\langle\, B, d \,\rangle) \sqsubseteq^\sharp \langle\, B^\sharp, d^\sharp \,\rangle \end{cases}$$

The following proposition is proved by induction on the number of transitions.

Proposition 11 (c-computations correctness).
For every concrete c-computation $s = \{\langle\, A_i, c_i \,\rangle\}_{i<|s|}$ of P there exists a corresponding abstract $\alpha(:\{c\}:)$-computation $s^\sharp = \{\langle\, A^\sharp_i, c^\sharp_i \,\rangle\}_{i<|s^\sharp|}$ of $\alpha(P)$ such that $|s| = |s^\sharp|$ and for all $0 \le i < |s|$ we have $\alpha(\langle\, A_i, c_i \,\rangle) \sqsubseteq^\sharp \langle\, A^\sharp_i, c^\sharp_i \,\rangle$.

Corollary 12 (c.a.c. correctness).
$\alpha(\,{\downarrow}(\mathcal{O}[\![\, D.A \,]\!] c)) \sqsubseteq^\sharp \bigsqcup^\sharp (\mathcal{O}[\![\, \alpha(D.A) \,]\!] \alpha(:\{c\}:))$

Note that in general the converse of Lemma 10 does not hold; in particular the concrete program may suspend while the abstract one has a transition. As a consequence, a finite concrete computation can be mapped into a diverging abstract computation, i.e. this approximation of the semantics does not preserve the termination's modes. Nonetheless, Proposition 11 ensures that every *finite* concrete computation is correctly approximated by a *partial* abstract computation of the same finite length.

Definition 7 and 8 do not require that the abstract domain is a constraint system and neither that it can be obtained as the Hoare's powerdomain of a constraint system. In the latter case we are in an *ideal situation* where a simpler notion of correctness can be used instead.

Definition 13.
An abstract constraint system $\mathcal{A}^{\top} = \langle L, \dashv^\sharp, \bot^\sharp, \top^\sharp, \otimes^\sharp, \sqcap^\sharp, V, \exists^\sharp_x, d^\sharp_{xy} \rangle$ is *correct* wrt the constraint system $\mathcal{C} = \langle C, \dashv, true, \otimes, \sqcap, V, \exists_x, d_{xy} \rangle$, using a surjective and monotonic function $\alpha : C \to L$, iff for each $c, d \in C$ (s.t. $c \otimes d \in C$), $x, y \in V$

$$\begin{array}{ll} \alpha(c \otimes d) & \dashv^\sharp \ \alpha(c) \otimes^\sharp \alpha(d) \\ \alpha(\exists_x c) & \dashv^\sharp \ \exists^\sharp_x \alpha(c) \\ \alpha(d_{xy}) & = \ d^\sharp_{xy} \end{array}$$

Let $\mathcal{A}^{\top}$ be an abstract constraint system which is correct wrt the constraint system $\mathcal{C}$ using α. Observe that $\otimes^\sharp$ is the *lub* over $\mathcal{A}^{\top}$.

Proposition 14.
1. $\mathcal{H}(\mathcal{A}^{\top})$ *is* down–correct *wrt* $\mathcal{H}(\mathcal{C})$ *using* $\tilde{\alpha}$ *(the additive extension of* α*)*
2. α *is a* $\otimes$*–morphism between* C *and* L
3. $\tilde{\alpha}$ *is a complete* $\tilde{\otimes}$*–morphism between* $\mathcal{P}{\downarrow}(C)$ *and* $\mathcal{P}{\downarrow}(L)$

Defining abstract domains based on correct abstract constraint systems is a very difficult task. The previous proposition gives an explanation to this assertion: these domains have to satisfy properties that usually are too strong.

6 Examples

As a first example, we present the (somehow trivial) abstract constraint system of *untouched variables*[6] $\mathcal{V} = \langle \mathcal{P}(V), \subseteq, \emptyset, V, \otimes^\sharp, \cap, V, \exists^\sharp_x, d^\sharp_{xy} \rangle$, where

$$S \otimes^\sharp T = S \cup T \qquad\qquad d^\sharp_{xy} = \begin{cases} \{x, y\} & \text{if } x \not\equiv y \\ \emptyset & \text{otherwise} \end{cases}$$
$$\exists^\sharp_x S \ = S \setminus \{x\}$$

Let us assume that $\mathcal{C}$ is a concrete constraint system having variables in V and satisfying the following axiom [12]: $\forall c, d \in C \ . \ \exists_x c \vdash d \ \Rightarrow \ \exists_x d = d$. Note that even if this axiom is not a consequence of Definition 2, it is true in almost all the "real" constraint systems.

[6] To our knowledge, this domain has been firstly introduced in [15].

Proposition15. *Let $\alpha : C \to \mathcal{P}(V)$ being defined as $\alpha(c) = \{x \in V \mid \exists_x c \neq c\}$. The abstract constraint system $\mathcal{V}$ is correct wrt $\mathcal{C}$ by using α.*

Therefore, we are in the ideal situation of Definition 13 and we can define our abstract domain as the Hoare's powerdomain of $\mathcal{V}$. Having stated correctness, we can approximate every concrete ask evaluation (i.e. entailment check) by the corresponding abstract ask evaluation. Whenever the abstract computation suspends, we definitely know that every concrete constraint described by the abstract store contains no information about one (or more) of the variables touched by the ask. As a consequence all the associated concrete computations will suspend too and we are safe.

Remark. This abstract domain is very weak: every time we perform an abstract procedure call we lose all the information about the actual parameter. This is due to the interaction between the abstract cylindric operator and the abstract diagonal element, namely when performing the parameter passing we compute $\exists^\sharp_x d^\sharp_{xy} = \{y\}$. As a consequence the usefulness of this domain is restricted to local (i.e. intra-procedural) analyses. However the solution of such a problem is well known: we have to consider a richer abstract domain (e.g. one of the domains for freeness analysis given in the literature), where also some information about *variable sharing* is taken into account.

6.1 Abstracting the constraint system $\mathcal{R}_{LinEq}$

Even if previous example is not very involved, the same approach is valid for any admissible downward closed property of any constraint system. Some examples of this kind of abstract domains can be found in the literature.

[13] describes an abstract domain for the static analysis of `CLP` programs that is useful for the detection of *definitely free* variables in the presence of both Herbrand constraints as well as systems of linear equations. Let us consider the latter case. Given a linear equation system

$$E = \left\{ \begin{array}{l} a_{11}X_1 + a_{12}X_2 + \ldots + a_{1n}X_n = b_1 \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{m1}X_1 + a_{m2}X_2 + \ldots + a_{mn}X_n = b_m \end{array} \right.$$

where $X_1, \ldots, X_n$ are variables and a_{ij} and b_j are numbers, variable X_i is definitely free if there does not exist a linear combination of the equations in E having the form $X_i = b$. Denoting $lc(E)$ the infinite set of linear combinations of equations in E, [13] defines the following abstraction function.

$$\alpha(E) = \left\{ \{X_1, \ldots, X_k\} \;\middle|\; \begin{array}{ll} (a_1X_1 + \ldots + a_kX_k = b) \in lc(E) & \\ a_i \neq 0 \qquad i = 1, \ldots, k & k > 0 \end{array} \right\}$$

Thus, the abstract domain is $\mathcal{A} = \langle\, \mathcal{P}(\mathcal{P}(V)\backslash\emptyset), \subseteq, \cup, \cap, \otimes^\sharp, V, \exists^\sharp_x, d^\sharp_{xy} \,\rangle$ where

$$S_1 \otimes^\sharp S_2 = S_1 \cup S_2 \cup \left\{ A \;\middle|\; \begin{array}{ll} A = (A_1 \cup A_2)\setminus D, & A \neq \emptyset \\ A_1 \in S_1,\ A_2 \in S_2,\ D \subseteq A_1 \cap A_2 & \end{array} \right\}$$

$$\exists^\sharp_x S = \{A \in S \mid x \notin A\} \qquad d^\sharp_{xy} = \left\{ \begin{array}{ll} \{\{x, y\}\} & \text{if } x \not\equiv y \\ \emptyset & \text{otherwise} \end{array} \right.$$

We refer to [13] for a complete definition of the domain and for the proofs of the abstract operators' correctness. Intuitively, the correctness conditions ensure that all the possible linear combinations of concrete equations are described by the computed abstract element. The abstract entailment is a containment test; as a particular case, if the abstract linear combination $\{X_i\}$ is not a member of the abstract store description, we can safely say that variable X_i is free.

In [13] it is also shown that inequalities and disequations can be correctly abstracted in the same way, namely by reading them as equations.

Example 1. Consider this definition of the process `length`, computing the length of a list, together with the following initial configuration.

```
length(L,N) :-
  ask(L=[]) -> tell(N=0)
    +
  ask(L=[_|_]) -> ∃L1,N1 in
                  tell(L=[_|L1],N=N1+1) || length(L1,N1).

⟨ produce(L) || length(L,N) || ask(N>20) -> consume(L) , true ⟩
```

By substituting each concrete constraint by its abstraction, we obtain the corresponding abstract program and initial agent; note that Herbrand constraints are mapped into the least abstract element $\emptyset$, while the disequation `N>20` is treated as `N=20`.

```
length(L,N) :-
  ask(∅) -> tell({{N}})
    +
  ask(∅) -> ∃L1,N1 in tell({{N,N1}}) || length(L1,N1).

⟨ produce(L) || length(L,N) || ask({{N}}) -> consume(L) , ∅ ⟩
```

By considering the possible abstract transitions of this program, we can easily make the following observations.

1. The abstract ask guard associated to the process `consume` is initially suspended (i.e. the abstract entailment test $\emptyset \supseteq \{\{N\}\}$ is not satisfied);
2. as long as we reduce the process `length` by selecting the *second* branch of its definition, the abstract global store will not change; namely we compute the store $\exists^{\sharp}_{N1}\{\{N, N1\}\} = \emptyset$, going back to the initial situation; therefore the ask guard associated to the process `consume` keeps suspending;
3. when reducing the process `length` by selecting the *first* branch of its definition, the global store changes to $\{\{N\}\}$ and the abstract ask synchronization succeeds.

Therefore, processes `produce` and `length` can be executed concurrently, while process `consume` has to wait for the process `length` to reach the end of the list L, i.e. to terminate. Indeed, this is actually what happens in any concrete computation. Consider the sequence of *concrete* constraints c_i obtained by restricting on

variable N the stores generated by the process `length`. Note that $|L| > i$ implies $c_i \equiv (\exists_M N = M + i)$ but, since the variable M is unconstrained, c_i cannot entail the concrete guard `N>20`. This behaviour is not very satisfactory and probably it does not correspond to our intended semantics, as we could prefer a situation where all of these processes can execute concurrently.

To conclude, in this example our approximation is able to detect an *undesired data dependency* between the process `length` and the ask guard associated to `consume`[7].

7 Toward an abstract semantics

In this section we will informally consider the problems related to the construction of an abstract semantics that correctly approximates the concrete one in the case of downward closed observables.

It is known that, in the general case, the c.a.c. semantics of a `CC` program is not invariant wrt different schedulings of parallel processes, i.e. it is not *confluent*. In principle, confluence is not needed to correctly define a static analysis framework. However, in order to be really useful, a static analysis must be correct wrt all the possible scheduling and must not be too inefficient. Therefore, when considering real programs, confluence becomes as desirable as correctness [15]. As a matter of fact, almost all the literature concerning the static analysis of `CC` languages [4, 5, 14, 15, 28] considers a two-steps approximation; in the first step the standard semantics is replaced by a confluent non-standard semantics, which is then abstracted in the second step. These intermediate semantics are correct wrt the standard one, but usually *must pay* in terms of accuracy of the results.

This is not the case when considering downward closed properties, because we can base our static analysis on a confluent semantics being as precise as the c.a.c. semantics. Confluence is easily obtained by reading the `CC` indeterministic program as a nondeterministic program (an *angelic* program, using the terminology of [18]), that is by interpreting all the *don't care* choice operators of the program as *don't know* choice operators. In the nondeterministic case, when considering a choice operator we split the control and consider all the branches. In the transition system this difference is captured by replacing rule **R4** of Table 2 with the following.

$$\textbf{R4}'\ \frac{d \vdash c}{\langle\, \texttt{ask(}c\texttt{)->}A, d \,\rangle \longrightarrow \langle\, A, d \,\rangle} \qquad \textbf{R4}''\ \frac{j \in \{1, \ldots, n\}}{\langle\, \sum_{i=1}^{n} A_i, d \,\rangle \longrightarrow \langle\, A_j, d \,\rangle}$$

Observe that the only difference between the two programs is that the indeterministic program has less suspensions; however, due to the monotonic nature of `CC` computations, for every suspended computation of the nondeterministic

[7] This data dependency can be avoided by telling the constraint `N1>=0` (or equivalently `N>0`) in the second branch of the definition of the process `length`.

program there exists a (terminated or suspended or infinite) computation in the original program that computes a stronger store. Let $\mathcal{O}'$ be the c.a.c. semantics based on the confluent transition system.

Proposition 16. *For all* $c \in C$. $\downarrow(\mathcal{O}[\![P]\!]c) = \downarrow(\mathcal{O}'[\![P]\!]c)$.

Technical problems related to termination can be solved essentially in the same way as it was done in [4].

Let us now consider some of the *denotational* semantics proposed in the literature. In [26] deterministic cc processes are elegantly modelled as upper closure operators (uco's) over the constraint system. The main property of this kind of representation is that any uco is fully determined by the set of its fixpoints. Moreover all the semantic operators on processes are naturally mapped into simple set theoretic operations over their representations, e.g. the parallel composition of two processes is obtained by intersecting their sets of fixpoints. [18] extends such a semantics to nondeterministic cc languages. When upward closed observables are considered, each (nondeterministic) process can be mapped into a *linear* uco over the Smyth's powerdomain of the constraint system. These functions can be coded as sets of (singleton) fixed point, essentially in the same way as it was done in [26] for the deterministic case. In [18] it is also shown that these processes can be alternatively modelled as *sets* of uco's on the (simple) constraint system. Different sets of closure operators may in general denote the same nondeterministic process and therefore the Smyth's powerdomain construction is applied onto the (extensionally ordered) domain of uco's. Such an alternative semantics definition can be easily adapted to model the abstract case, provided that we are dealing with an abstract constraint system (see Definition 13). However, as we are observing downward closed properties, we should consider the Hoare's powerdomain of the domain of uco's.

Definition 17. The Hoare's powerdomain of uco's on the constraint system C is

$$\mathcal{H}(uco(C)) = \langle\, \mathcal{P}{\downarrow}(uco(C)), \subseteq, \bot_H, \top_H, \uplus, \cap \,\rangle$$

where $\mathcal{P}{\downarrow}(uco(C))$ is the set of all the non–empty subsets of uco's on C that are downward closed and admissible wrt the extensional ordering; $\{\!|\cdot|\!\} : uco(C) \to \mathcal{P}{\downarrow}(uco(C))$ is defined as $\{\!|f|\!\} = \{\, g \in uco(C) \mid \forall c \in C \,.\, g(c) \dashv f(c) \,\}$; $\uplus$ is the closure of the union, $\bot_H = \{\!|C|\!\} = \{id\}$ is the bottom element and $\top_H = uco(C)$ is the top element.

The equations modelling the semantic functions look essentially the same as those given in [18] (see Table 3, where Π is the set of process names and $\Im = \Pi \to \mathcal{P}{\downarrow}(uco(C))$ is the domain of environments). The only difference is the definition of the singleton embedding operator.

Unfortunately, most of the abstract domains modelling downward closed properties are not constraint systems. In these cases, if we are interested in a denotational abstract semantic construction, we can consider a suitable variant of the approach based on ask/tell traces developed in [10, 11]. Here the first

$$\mathcal{N} : \text{Progr} \rightarrow \mathcal{P}\downarrow(uco(C))$$

$$\mathcal{N}[\![D.\ A]\!] = \mathcal{E}[\![A]\!](lfp\, \mathcal{D}[\![D]\!])$$

$$\mathcal{D} : \text{Dec} \times \Im \rightarrow \Im$$

$$\mathcal{D}[\![\epsilon]\!] I = I$$

$$\mathcal{D}[\![\mathtt{p(x)}\text{:-}A.\ D]\!] I = \mathcal{D}[\![D]\!](I[\mathtt{p} \mapsto \mathcal{E}[\![\exists\, \mathtt{x}\ \mathtt{in}\, (\mathtt{tell}(d_{x\mu}) \parallel A)]\!] I])$$

$$\mathcal{E}[\![\cdot]\!] : \text{Agent} \times \Im \rightarrow \mathcal{P}\downarrow(uco(C))$$

$$\mathcal{E}[\![\mathtt{Stop}]\!] I = \{|C|\}$$

$$\mathcal{E}[\![\mathtt{tell}\,(c)]\!] I = \{| \uparrow c |\}$$

$$\mathcal{E}[\![\mathtt{ask}\,(c)\text{->}A]\!] I = \biguplus \left\{ \{| \overline{\uparrow c} \cup (\uparrow c \cap f) |\} \;\middle|\; f \in \mathcal{E}[\![A]\!] I \right\}$$

$$\mathcal{E}[\![\exists\, \mathtt{x}\ \mathtt{in}\, A]\!] I = \biguplus \left\{ \{| \exists_x f |\} \;\middle|\; f \in \mathcal{E}[\![A]\!] I \right\}$$

$$\mathcal{E}[\![A \parallel B]\!] I = \biguplus \left\{ \{| f \cap g |\} \;\middle|\; f \in \mathcal{E}[\![A]\!] I\ ,\ g \in \mathcal{E}[\![B]\!] I \right\}$$

$$\mathcal{E}[\![\sum_{i=1}^{n} A_i]\!] I = \biguplus_{i=1}^{n} \mathcal{E}[\![A_i]\!] I$$

$$\mathcal{E}[\![\mathtt{p(y)}]\!] I = \biguplus \left\{ \{| \exists_\mu (\uparrow d_{\mu y} \cap f) |\} \;\middle|\; f \in I[p] \right\}$$

where

$$\exists_x f = \{ d \in C \mid \exists_x d = \exists_x c,\ c \in f \}$$

Table 3. The generalized semantics

problem to solve is termination, because a trace can be infinite even if it is defined over a finite abstract domain. We think that a notion of *canonical form* for traces (similar to the one developed in [26] for bounded trace operators) would suffice.

It is worth pointing out that the approximation theory developed in this work can be applied to any kind of semantic construction dealing with the basic mechanism of blocking ask. As a matter of fact, note that we already proved the correctness result for (the abstract version of) the c-computations semantics (see Proposition 11), which observes all the intermediate steps of the concrete computations; we also implicitly used this semantics in Example 1. Therefore,

our technique can be also applied to semantics observing *the way* the answer constraints are actually computed. We believe that such a correctness result can be easily lifted to the case of a non-interleaving semantics, e.g. by considering a variant of the *true concurrent* semantics developed in [21, 22]. In this case the definite suspension information could be useful to obtain upper bounds to the degree of parallelism of the program.

8 Conclusions and related works

The static analysis of CC languages is a relatively new but very active area of research. To our knowledge, this is the first work on this topic in which a domain independent correct approximation of ask constraints is identified. Almost all the previous works about the static analysis of CC programs [4, 14, 15, 28] either consider a specific constraint system or *assume* that a correct ask approximation has already been found. [5] claims that it is possible to abstract ask constraints in a domain independent way even when considering entailment closed properties (e.g. groundness). This result contrasts with a *negative* result established in [28] and simple counterexamples can be shown that prove the uncorrectness of such an approach in the general case.

The approximation described in our work can be applied to a wide class of program properties, namely the downward closed ones. Several properties falling in this class (e.g. freeness) have already been studied in the context of the static analysis of sequential (constraint) logic languages. In our opinion the same abstract domains can be used in the CC case, provided that a suitable abstract semantic construction is identified. At the same time, we strongly believe that such a general result can motivate the study of "new" downward closed properties. This approximation theory allows to detect definitely suspended branches of the computation. Such an information can be usefully applied to the debugging and specialization of CC programs. Another area of application could be the compile-time (partial) scheduling of concurrent processes; whenever our analysis can prove that two or more processes are definitely serialized, we can avoid their costly and harmful parallel execution.

The definition of "the right" abstract semantics is an open problem. We have shown that if we are only interested in the downward closed properties obtainable from the c.a.c. semantics, then we can assume that all the choice operators in our program are local, thus achieving the confluence of the computation without any loss of precision. In our opinion, however, an extensive study of the cost/precision tradeoffs of the different abstract semantics proposals is strongly needed.

It has been recently shown that *delay* mechanisms in both sequential constraint languages and constraint solvers can be formalized as asks primitives on an underling domain (see [1] for the definition of ask&tell constraint systems and some related issues). This connection is currently being further investigated from the point of view of ask approximation.

Acknowledgements: The author would like to thank Catuscia Palamidessi for her valuable comments and suggestions on a previous version of this work.

References

1. R. Bagnara. Constraint Systems for Pattern Analysis of Constraint Logic–Based Languages. Presented at the *First Int'l Workshop on Concurrent Constraint Programming*, Venice, Italy, 1995.
2. G. Birkhoff. Lattice Theory. In *AMS Colloquium Publication, third ed.*, 1967.
3. M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness Analysis for Logic Programs - And Correctness? In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 116–131. The MIT Press, Cambridge, Mass., 1993.
4. M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proc. of the 20th International Colloquium on Automata, Languages, and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 633–644, 1993.
5. C. Codognet and P. Codognet. A general semantics for Concurrent Constraint Languages and their Abstract Interpretation. In M. Meyer, editor, *Workshop on Constraint Processing at the International Congress on Computer Systems and Applied Mathematics, CSAM'93*, 1993.
6. A. Cortesi and G. Filè. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *Proc. ACM Symposium on Partial Evaluation and Semantics-based Program Transformation*, pages 52–61. ACM Press, 1991.
7. 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.
8. P. Cousot and R. Cousot. A constructive characterization of the lattices of all retracts, pre-closure, quasi-closure and closure operators on a complete lattice. *Portugaliæ Mathematica*, 38(2):185–198, 1979.
9. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
10. F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer-Verlag, Berlin, 1991.
11. F.S. de Boer and C. Palamidessi. A process algebra for concurrent constraint programming. In K. Apt, editor, *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, Series in Logic Programming, pages 463–477, Washington, USA, 1992. The MIT Press, Cambridge, Mass.
12. F.S. de Boer, C. Palamidessi, and A. Di Pierro. Infinite Computations in Nondeterministic Constraint Programming. *Theoretical Computer Science*. To appear.
13. V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 100–115. The MIT Press, Cambridge, Mass., 1993.
14. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proc. of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221. IEEE Computer Society Press, 1993.

15. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. In *Proc. of the Fourth International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, Montreal, Canada, 1995.
16. R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.
17. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.
18. R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Science Lab., Xerox PARC, 1991.
19. M. Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Netherlands, 1992. REX Workshop, Springer-Verlag, Berlin.
20. K. Marriott and P. J. Stuckey. Approximating Interaction between Linear Arithmetic Constraints. In M. Bruynooghe, editor, *Proc. 1994 Int'l Logic Programming Symposium*, pages 571–585. The MIT Press, Cambridge, Mass., 1994.
21. U. Montanari and F. Rossi. Contextual Occurrence Nets and Concurrent Constraint Programming. In *Proc. Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.
22. U. Montanari and F. Rossi. A Concurrent Semantics for Concurrent Constraint Programming via Contextual Nets. In *Principles and Practice of Constraint Programming*. The MIT Press, Cambridge, Mass., 1995.
23. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freness of Program Variables through Abstract Interpretation. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 49–63. The MIT Press, Cambridge, Mass., 1991.
24. G.D. Plotkin. Pisa lecture notes. Unpublished notes, 1981-82.
25. V. A. Saraswat, K. Kahn, and J. Levy. `Janus`: A step towards distributed constraint programming. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 431–446. The MIT Press, Cambridge, Mass., 1990.
26. 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.
27. 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.
28. E. Zaffanella, G. Levi, and R. Giacobazzi. Abstracting Synchronization in Concurrent Constraint Programming. In M. Hermenegildo and J. Penjam, editors, *Proc. Sixth Int'l Symp. on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 57–72. Springer-Verlag, 1994.

This article was processed using the LaTeX macro package with LLNCS style