Domain Independent Ask Approximation in CCP*

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

Abstract

The main difficulty in the definition 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* observations. Properties falling in this class (e.g. freeness) have already been studied in the context of the analysis of sequential logic programs. We believe that the same abstract domains can be usefully applied to the CC context to provide meaningful ask approximations.

1 Introduction

Concurrent Constraint (CC) programming [16] 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*, a constraint representing the global state of the computation. Communication 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 observations. As a consequence, the approximation theory developed to correctly characterize *upward closed* (i.e. closed wrt entailment) properties becomes useless when we are looking for a domain independent solution to the ask approximation problem [18].

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

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

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

Table 1: The syntax

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. debugging of `CC` programs as well as identifying processes that are definitely serialized (so that we avoid their harmful parallel execution). However its usefulness is first of all in the improvement of 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 observations is not new. See [12] for an interesting discussion about *safety* and *liveness* properties, 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 observations, e.g. freeness in the Herbrand as well as in arithmetic constraint systems [6]. It is our opinion that these same abstract domains can be usefully applied to the `CC` context and provide meaningful ask approximations.

2 The language

`CC` is not a language, it is a class of languages parametric wrt the underlying constraint system. In [16] constraint systems are defined by enclosing typical cylindric algebra's operators (cylindrifications and diagonal elements [10]) in the well known formalization of *partial information systems* [17], which model the gathering and the management of a set of elementary assertions by means of a compact entailment relation. We refer to [16] for a more detailed presentation.

Definition 2.1
A (cylindric) *constraint system* $\mathcal{C}^{\top} = \langle\, C \cup \{\mathit{false}\}, \dashv, \mathit{true}, \mathit{false}, \otimes, \sqcap, V, \exists_x, d_{xy}\,\rangle$ is an algebraic structure where

- $\langle\, C, \dashv, \mathit{true}, \otimes, \sqcap\,\rangle$ is a partial information 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. $\exists_x \mathit{false} = \mathit{false}$
  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. $d_{xx} = \mathit{true}$
  2. $z \neq x, y$ implies $d_{xy} = \exists_z (d_{xz} \otimes d_{zy})$
  3. $x \neq y$ 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 described by a transition system $T = (\mathit{Conf}, \longrightarrow)$. 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. $\exists\, \mathtt{x}\, \mathtt{in}\, A \equiv \exists (x, \mathit{true})\, \mathtt{in}\, A$. Parallelism is modeled 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 ask(c_i) is entailed by the store, i.e. $d \vdash c_i$. The guarded choice operator nondeterministically 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 [16], where $\mathtt{p(x)\!:\!-}A \in P$ and $\Delta^{\mathtt{y}}_{\mathtt{x}} A$ is defined as follows [5].

$$\Delta^{\mathtt{y}}_{\mathtt{x}} A \;=\; \begin{cases} A & \text{if } \mathtt{x} \equiv \mathtt{y} \\ \exists\, \mathtt{x}\, \mathtt{in}\, (\mathtt{tell}(d_{xy}) \parallel A) & \text{otherwise} \end{cases}$$

A c-computation s for a program $D.A$ is a possibly infinite and fair sequence of configurations $\langle A_i, c_i \rangle_{i<\omega}$ such that $A_0 = A$ and $c_0 = c$ and for all $i < |s|$, $\langle A_i, c_i \rangle \longrightarrow \langle A_{i+1}, c_{i+1} \rangle$.

R1	$\langle\, \mathtt{tell}(c), d \,\rangle \longrightarrow \langle\, \mathtt{Stop}, d \otimes c \,\rangle$
R2	$\dfrac{\langle\, A, c \otimes \exists_x d \,\rangle \longrightarrow \langle\, A', c' \,\rangle}{\langle\, \exists(x, c)\ \mathtt{in}\ A, d \,\rangle \longrightarrow \langle\, \exists(x, c')\ \mathtt{in}\ A', d \otimes \exists_x c' \,\rangle}$
R3	$\dfrac{\langle\, A, c \,\rangle \longrightarrow \langle\, A', d \,\rangle}{\begin{array}{c}\langle\, A \parallel B, c \,\rangle \longrightarrow \langle\, A' \parallel B, d \,\rangle \\ \langle\, B \parallel A, c \,\rangle \longrightarrow \langle\, B \parallel A', d \,\rangle\end{array}}$
R4	$\dfrac{j \in \{1, \ldots, n\} \ \wedge\ d \vdash c_j}{\langle\, \sum_{i=1}^{n} \mathtt{ask}(c_i)\mathtt{->}A_i, d \,\rangle \longrightarrow \langle\, A_j, d \,\rangle}$
R5	$\dfrac{\mathtt{p(x)\!:\!-}A \in P}{\langle\, \mathtt{p(y)}, d \,\rangle \longrightarrow \langle\, \Delta^{\mathtt{y}}_{\mathtt{x}} A, d \,\rangle}$

Table 2: The transition system T

Let $\not\longrightarrow$ denote the absence of admissible transitions. Computations reaching configuration $\langle\, A_n, c_n \,\rangle \not\longrightarrow$ are called *finite* computations and c_n is the (finite) computed answer constraint. 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 2.2 The semantics for program $P = D.A$ in the store c is

$$\begin{array}{rcl} \mathcal{O}[\![\, D.A \,]\!](c) & = & \left\{ d \in C \,\middle|\, \langle\, A\,,\, c \,\rangle \xrightarrow{*} \langle\, B\,,\, d \,\rangle \not\longrightarrow \right\} \\ & \cup & \left\{ d \in C \,\middle|\, \begin{array}{l} \langle\, A_0\,,\, c_0 \,\rangle \longrightarrow \ldots \longrightarrow \langle\, A_i\,,\, c_i \,\rangle \longrightarrow \ldots \\ A_0 = A\,,\ c_0 = c\,,\ d = c_0 \otimes \ldots \otimes c_i \otimes \ldots \end{array} \right\} \end{array}$$

Note that this semantics collects the limit constraints of infinite 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*.

3 Program properties and approximations

As we have seen, the operational 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 observations of the program are a *subset* of the property, i.e. $\mathcal{O}[\![\, P \,]\!](c) \subseteq \phi$. Following this general view, 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 [3] formalizes the approximation construction process by mapping concrete semantic objects and operators into corresponding abstract semantic objects and operators.

We write $\uparrow(\phi)$ to denote the *upward closure* of the program property ϕ, namely the set $\{c \in C \mid \exists\, d \in \phi \,.\, c \vdash d\}$; a property is upward closed iff it is equivalent to its upward closure, i.e. $\phi = \uparrow(\phi)$. Downward closed properties are defined dually. 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, *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 such that 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 observations. In [18] entailment closed[1] properties are considered. The main result is that it is impossible to develop a meaningful generalized semantics for CC languages in the style of [9], 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 [18].

Remark 3.1 *If ϕ is downward closed then $\mathcal{O}[\![P]\!](c) \subseteq \phi \;\Leftrightarrow \downarrow(\mathcal{O}[\![D]\!](c)) \subseteq \phi$.*

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 3.1 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

[1]Due to a dual definition of the ordering on the constraint system, in [18] 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 [3].

example of a property that is not admissible, consider the following definition of *non-groundness*: 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 this analysis, we have to redefine the property, e.g. by stating that if c binds a 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 [14, 17] construction over the constraint system characterizes this kind of observations.

Definition 3.2 The Hoare's powerdomain of the constraint system $\mathcal{C}$ is the complete lattice $\mathcal{H}(\mathcal{C}) = \langle \mathcal{P}{\downarrow}(C), \subseteq, \{true\}, C, \uplus, \cap \rangle$, 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 observations 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 also 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 observations 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 [15] for a discussion of this topic in distributed programming).
From now on $\tilde{\otimes}$ and $\tilde{\exists}_x$ will denote the extensions of $\otimes$ and $\exists_x$ over $\mathcal{H}(\mathcal{C})$.

- $\forall S, T \in \mathcal{P}{\downarrow}(C) \; . \; S \,\tilde{\otimes}\, T = \biguplus \left\{ :\{c \otimes d\}: \;\middle|\; c \in S,\, d \in T,\, c \otimes d \in C \right\}$
- $\forall S \in \mathcal{P}{\downarrow}(C) \; . \; \tilde{\exists}_x S = \biguplus \left\{ :\{\exists_x c\}: \;\middle|\; c \in S \right\}$

Note that the *merge over all paths* operator [3] is provided by the *lub* of $\mathcal{H}(\mathcal{C})$. Also note that in general $\tilde{\otimes}$ is not idempotent, while being extensive.

4 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. 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 4.1 An abstract domain $\mathcal{A} = \langle\, L, \sqsubseteq^\sharp, \perp^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp, \otimes^\sharp, V, \exists^\sharp{}_x, d^\sharp{}_{xy} \,\rangle$ 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 $\forall S, T \in \mathcal{P}{\downarrow}(C)$, $\forall x, y \in V$

1. $\mathcal{L} = \langle\, L, \sqsubseteq^\sharp, \perp^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \,\rangle$ is a complete lattice
2. there exists γ s.t. (α, γ) is an upper Galois insertion[2] relating $\mathcal{H}(\mathcal{C})$ and $\mathcal{A}$.
3. $\alpha(S \tilde{\otimes} T) \sqsubseteq^\sharp \alpha(S) \otimes^\sharp \alpha(T)$
4. $\alpha(\tilde{\exists}_x S) \sqsubseteq^\sharp \exists^\sharp{}_x \alpha(S)$
5. $\alpha(:\!\{d_{xy}\}\!:) \sqsubseteq^\sharp d^\sharp{}_{xy}$

From now on, we assume that the abstract domain $\mathcal{A}$ is *down–correct* wrt the constraint system $\mathcal{C}$ using α and prove that such a notion of correctness implies 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 $\sqsubseteq^\sharp$. To this end it is sufficient to consider the operational semantics.

Definition 4.2 Given the concrete agent A, the corresponding abstract agent $A^\sharp = \alpha(A)$ is obtained by replacing all the concrete constraints $c \in C$ occurring in A by the corresponding abstractions $c^\sharp = \alpha(:\!\{c\}\!:) \in L$.

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. Let A be an agent defined over the constraint system $\mathcal{C}$, let $c \in C$ be a concrete store and let $c^\sharp \in L$ be a description such that $\alpha(:\!\{c\}\!:) \sqsubseteq^\sharp c^\sharp$.

Lemma 4.1 (correctness)
$\langle\, A\,,\, c \,\rangle \longrightarrow \langle\, B\,,\, d \,\rangle$ *implies* $\langle\, \alpha(A)\,,\, c^\sharp \,\rangle \longrightarrow \langle\, \alpha(B)\,,\, d^\sharp \,\rangle$ *and* $\alpha(:\!\{d\}\!:) \sqsubseteq^\sharp d^\sharp$.

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

Proposition 4.2 *For every concrete c-computation of P yielding the constraint $d \in C$ there exists a corresponding abstract $\alpha(:\!\{c\}\!:)$-computation of $\alpha(P)$ yielding the description $d^\sharp$ such that $\alpha(:\!\{d\}\!:) \sqsubseteq^\sharp d^\sharp$.*

Note that in general the converse of Lemma 4.1 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 corresponding abstract infinite computation. Therefore, even in the case that we are interested in finite computations only, the abstract semantics *must* consider infinite computations in order to be correct.

[2] Given two complete lattices $\langle\, L, \leq \,\rangle$ and $\langle\, L', \leq' \,\rangle$, an *upper Galois connection* between L and L' is a pair of adjoint functions (α, γ) such that $\alpha : L \to L'$ and $\gamma : L' \to L$ and $\forall x \in L\,.\, \forall y \in L'\,.\, \alpha(x) \leq' y \Leftrightarrow x \leq \gamma(y)$. An upper Galois *insertion* between L and L' is an upper Galois connection such that α is surjective (equivalently, γ is one-to-one).

Definition 4.2 does 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 4.3
An abstract constraint system $\mathcal{A} = \langle L, \dashv^\sharp, \bot^\sharp, \top^\sharp, \otimes^\sharp, \sqcap^\sharp, V, \exists^\sharp{}_x, d^\sharp{}_{xy} \rangle$ is *correct* wrt the constraint system $\mathcal{C} = \langle C, \dashv, \mathit{true}, \otimes, \sqcap, V, \exists_x, d_{xy} \rangle$, using a surjective and monotonic function $\alpha : C \to D$, iff for each $c, d \in C$, $x, y \in V$

1. $\alpha(c \otimes d) \dashv^\sharp \alpha(c) \otimes^\sharp \alpha(d)$
2. $\alpha(\exists_x c) \dashv^\sharp \exists^\sharp{}_x \alpha(c)$
3. $\alpha(d_{xy}) = d^\sharp{}_{xy}$

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

Proposition 4.3

1. *$\mathcal{H}(\mathcal{A})$ is* down–correct *wrt $\mathcal{H}(\mathcal{C})$ using $\tilde{\alpha}$ (the additive extension of α)*
2. *α is a complete $\otimes$–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 of this assertion: these domains have to satisfy properties that usually are too strong.

4.1 A toy example

As a first example we present the abstract constraint system of *untouched variables*[3] $\mathcal{V} = \langle \mathcal{P}(V), \subseteq, \emptyset, V, \otimes', \cap, V, \exists'_x, d'_{xy} \rangle$, where

$$\begin{aligned} S \otimes' T &= S \cup T \\ \exists'_x S &= S \backslash \{x\} \end{aligned} \qquad\qquad d'_{xy} = \begin{cases} \{x, y\} & \text{if } x \not\equiv y \\ \emptyset & \text{otw.} \end{cases}$$

Let us assume that $\mathcal{C}$ is a concrete constraint system having variables in V and satisfying the following axiom [5]: $\forall c, d \in C \,.\, \exists_x c \vdash d \;\Rightarrow\; \exists_x d = d$. Note that even if this axiom is not a consequence of Definition 2.1, it is true in almost all the "real" constraint systems.

Proposition 4.4 *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.*

[3]To our knowledge, this domain has been firstly introduced in [8]. The formal definition of α was given to me by Catuscia Palamidessi, during an interesting discussion related to other topics.

Therefore, we just are in the ideal situation of Definition 4.3 and we can define our abstract domain as the Hoare's powerdomain of $\mathcal{V}$. Having proved correctness, we can approximate every concrete ask evaluation (i.e. entailment check) by the corresponding abstract ask evaluation. Let us see the intuition behind this result. Suppose the abstract ask evaluation does not succeed; this means that there exists a variable x occurring free in the concrete ask constraint such that x is definitely unbounded in all the concrete constraints described by the abstract store. As a consequence all the associated concrete computations will suspend too and we are safe.

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

Previous example seems just a toy. However, 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.

[6] 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

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

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 $X_i = n$. Denoting $lc(E)$ the infinite set of linear combinations of equations in E, they define the following abstraction function.

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

We refer to [6] for a complete definition of the domain and of the abstract operators. Intuitively, the correctness of the analysis ensures that all the possible linear combinations of concrete equations are described by the computed abstract element. 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. [6] also shows how to correctly deal with inequalities and disequations (i.e. the constraint system $\mathcal{R}^{\neq}_{Lin}$ is considered).

5 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 standard one in the case of downward closed observations.

In the general case, the observations of a `CC` program are not invariant wrt different schedulings of parallel processes, i.e. the operational semantics 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 programs being a little bit bigger than toy examples, confluence becomes as desirable as correctness [8]. As a matter of fact, almost all the literature concerning the static analysis of CC languages considers non–standard semantics that are confluent [1, 2, 7, 8, 18]. These 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 standard one. Confluence is easily obtained by reading the CC indeterministic program as if it were an *angelic* program [11], that is by interpreting all the *don't care* choice operators of the program as *don't know* choice operators. In the angelic case, when considering a choice operator we split the control and consider all the branches. In the operational semantics this difference is captured by replacing rule **R4** in Table 2 with the following.

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

Observe that the only difference between the two programs is that the original program has less suspensions; however, due to the monotonic nature of CC computations, for every suspended computation of the angelic program there exists a (terminated or suspended or infinite) computation in the original program that computes a stronger store. Let $\mathcal{O}'$ be the operational semantics based on the confluent transition system.

Proposition 5.1 *For all* $c \in C\,.\ {\downarrow}(\mathcal{O}[\![\, P\,]\!](c)) = {\downarrow}(\mathcal{O}'[\![\, P\,]\!](c))$.

Thus a first proposal of an abstract semantic construction can be based on the confluent transition system operational semantics. Technical problems related to termination can be solved essentially in the same way as it was done in [1].

In [16] it is shown how to elegantly model a deterministic CC process as an upper closure operator (uco), i.e. a monotonic, extensive and idempotent function 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 operators over their representations, e.g. the parallel composition of two processes is obtained by taking the intersection of their fixpoints' sets. [11] study the extension of such a semantics on angelic CC languages, where only local choice operators are allowed and upward closed observations are considered.

If the abstract domain we are dealing with is based on an abstract constraint system (see Definition 4.3) we are in a position to develop a semantic construction very similar to the latter. It is worth noting that, in such a semantic construction, the *process restartability* property is assumed. This property holds for deterministic programs [16] and it also holds for angelic programs when we consider upward closed observations [11], but it does not hold in the general case. However, when considering downward closed properties, it can be proved that correctness is still granted, while we pay something in the approximation's precision.

Unfortunately, many interesting 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 [4]. Here the first problem to solve is termination, because a trace can be infinite even if defined over a finite abstract domain. We think that a notion of *canonical form* for traces (similar to the one developed in [16]) 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. Therefore, even if all the semantics mentioned above only observe *the results* of a `CC` program, our technique can be also applied to semantics observing *the way* these results are actually computed. As an example, if we consider the *true concurrency* semantics developed in [13], the definite suspension information could be useful to obtain upper bounds to the degree of parallelism of a program or to discover undesired data dependencies between concurrent processes.

6 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 it is identified a domain independent correct approximation of ask constraints. Almost all the previous works about the static analysis of `CC` programs [1, 7, 8, 18] either consider a specific constraint system or *assume* that a correct ask approximation has already been found. In [2] a different kind of domain independent ask approximation has been considered. In our opinion, however, this framework requires the satisfaction of too strong correctness conditions and cannot be widely used.

The approximation described in the current work allows to detect definitely suspended branches of the computation and it may be therefore useful in the debugging and specialization of `CC` programs. It can be applied to a wide class of program properties, namely the downward closed ones. Some property falling in this class (e.g. freeness) has 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 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.

The definition of a suitable abstract semantics for the static analysis of this class of properties is an open problem. We have shown that if we are interested in downward closed properties only then we can assume that all the choice operators in our program are local, 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.

References

[1] 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.

[2] 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.

[3] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[4] 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.

[5] F.S. de Boer, C. Palamidessi, and A. Di Pierro. Infinite Computations in Nondeterministic Constraint Programming. *Theoretical Computer Science*. To appear.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.

[11] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Science Lab., Xerox PARC, 1991.

[12] 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.

[13] 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.

[14] G.D. Plotkin. Pisa lecture notes. Unpublished notes, 1981-82.

[15] 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.

[16] 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.

[17] 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.

[18] 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.