Abstracting Synchronization in Concurrent Constraint Programming *

Enea Zaffanella[1] Roberto Giacobazzi[2] Giorgio Levi[1]

[1] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa
(zaffanel,levi)@di.unipi.it

[2] LIX, Laboratoire d'Informatique, École Polytechnique
91128 Palaiseau cedex
giaco@lix.polytechnique.fr

Abstract. Because of synchronization based on *blocking ask*, some of the most important techniques for data flow analysis of (sequential) constraint logic programs (*clp*) are no longer applicable to *cc* languages. In particular, the *generalized* approach to the semantics, intended to factorize the (standard) semantics so as to make explicit the domain-dependent features (i.e. operators and semantic objects which may be influenced by abstraction) becomes useless for relevant applications. A possible solution to this problem is based on a more abstract (non-standard) semantics: the *success semantics*, which models non suspended computations only. With a program transformation (*NoSynch*) that simply ignores synchronization, we obtain a *clp*-like program which allows us to apply standard techniques for data flow analysis. For suspension-free programs the success semantics is equivalent to the standard semantics thus justifying the use of suspension analysis to generate sound approximations. A second transformation (*Angel*) is introduced, applying a different abstraction of synchronization in possibly suspending programs and resulting in a framework which is adequate to suspension analysis. Applicability and accuracy of these solutions are investigated.

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.

* The work of E. Zaffanella and G. Levi has been supported by the "PARFORCE" (Parallel Formal Computing Environment) BRA-Esprit II Project n. 6707. The work of R. Giacobazzi has been partly supported by the EEC *Human Capital and Mobility* individual grant: "Semantic Definitions, Abstract Interpretation and Constraint Reasoning", N. ERB4001GT930817, and by the Project inter-PRC: "Langages Logiques Concurrents avec Contraintes".

In the classical framework of abstract interpretation [5], 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 definition of an abstract interpreter for a language actually corresponds to a semantics abstraction. However, many aspects of the (concrete) semantic construction are not affected by the abstraction. The *generalized* approach to the semantics in [11] has been introduced to factorize the semantics with respect to its domain-dependent features (i.e. operators and semantic objects). This technique can be naturally applied to *clp* programs, where the notion of *constraint system* provides a uniform framework to deal with semantic objects (constraints) and operators at different levels of abstraction. In this case, abstract interpretation is obtained simply by evaluating the abstract program into an instance of *clp*, provided with a suitable *abstract constraint system*. The key issue here is that both concrete and abstract computations are instances, at the constraint system level, of the *clp* paradigm. In general, the abstraction is characterized by weakening constraints.

In this paper we extend the generalized semantics approach to the abstract interpretation of *cc* programs, and show that in general we cannot provide any correct approximation by evaluating an abstract version of the program. The *ask-tell* paradigm [16] is an extension of constraint logic programming: in addition to satisfiability (tell), *entailment* (ask) is introduced. This different view of constraint programming leads to a powerful paradigm for concurrent computations: *concurrent constraint programming* (*cc*) in a shared *store* [16]. A store is a constraint representing the global state of the computation. Synchronization is achieved through *blocking ask*. This mechanism introduces some problems when dealing with abstraction. Intuitively a correct approximation of the program meaning generates weaker answers for any possible program behaviour. Thus, in order to correctly characterize answers associated with suspended computations, we must guarantee that whenever a concrete computation suspends the corresponding abstract computation suspends too. This can only be obtained by replacing ask constraints with stronger constraints, which is usually not the case in abstract interpretation. To overcome this "negative" result we consider a more abstract semantics modeling non suspended computations only. A transformation which ignores synchronization can be applied to make applicable the generalized semantics approach to the static analysis of *cc* programs. For suspension-free programs the standard and the success semantics are equivalent, thus justifying the use of suspension analysis [2] to generate sound approximations.

A different schema can be obtained by introducing hybrid primitives to deal with ask constraints. As before, we use a program transformation (*Angel*) which essentially replaces *don't care* nondeterminism with *don't know* nondeterminism. Following the semantic characterization of angelic *cc* processes given in [14], we obtain the denotational counterpart of the transition system based suspension analysis in [2] (modulo the absence of the consistency check). Simple results relate the accuracy of these different solutions when the program is suspension-free, showing that the first approach is always better than the second one.

2 Constraint Systems

The algebraic specification (for sequential constraint logic programs) given in [11] is of major interest for abstract interpretation as it defines the minimal properties such a structure has to satisfy in order to obtain a suitable base for the generalized semantic construction. The resulting domains are very weak, allowing non-commutative and non-idempotent constraint composition operators and a wide range of (possibly non-distributive) constraint disjunction operators, i.e. widenings. On the other hand, the denotational semantics construction in [16] for *cc* languages requires stronger domains (only commutative and idempotent constraint composition operators are allowed). In this case constraint systems are not required to have a disjunction operator. Disjunctions arise only when considering different execution paths and they are modeled at the program semantics level (i.e. outside the constraint system definition) using sets of possible behaviours or a (fixed) powerdomain construction. As a consequence, these structures can be seen as specific instances of the previous ones (with minor modifications). Because of its specificity to the *cc* case, in the following we consider the latter approach.

The construction in [16] is an extension of Scott's partial information systems [17]. Informally, we have a denumerable set D of elementary assertions (containing distinct elements 1 and 0 representing the least informative assertion and the contradiction respectively) and a compact entailment relation $\vdash \subseteq \mathcal{P}^f(D) \times D$. By taking the entailment closure[3] $\delta(u)$ of a set of assertions u we obtain the equivalence relation $\sim$ ($u \sim v$ iff $\delta(u) = \delta(v)$). Hence, a *simple constraint system* is $C = \langle \mathcal{P}(D), \dashv \rangle / _\sim$, which is a complete ω-algebraic lattice [17]. An arbitrary element of C is called a *constraint*. Compact elements are called *finite constraints*, since they are equivalent to a finite subset of D. Finite constraints form the *base* B_C of the constraint system. In order to treat the hiding operator, [16] introduces a family of unary operations called *cylindrifications* [13]. Intuitively, given a constraint c, the cylindrification operation $\exists_x(c)$ yields the constraint obtained by "projecting out" information about the variable x from c. *Diagonal elements* [13] are considered as a way to provide parameter passing. Note that special variables (not accessible to the user) together with a suitable use of cylindrification and diagonal elements make variable renaming no longer needed.

Definition 1. A *(cylindric) constraint system*[4] $\langle C, \vdash, \mathit{false}, \mathit{true}, \otimes, V, \exists_x, d_{xy} \rangle$ is an algebraic structure where: $\langle C, \dashv \rangle$ is a simple constraint system, $\mathit{true} = [1]_\sim$ and $\mathit{false} = [0]_\sim$, $\otimes$ is the *glb*, V is a denumerable set of variables and $\forall x, y \in V$, $\forall c, d \in C$, the operator $\exists_x : C \to C$ satisfies

[3] The entailment closed representation, when it is finite, is a domain independent strong normal form for constraints and it is very useful when there are not simpler ones (e.g. $clp(\mathcal{FD})$). However, many domains do have a simpler strong normal form (Herbrand, *Prop*, *Sharing*, etc.) which greatly simplifies their representation.

[4] In order to have a standard approach when dealing with abstract interpretation, we order constraints in a dual fashion w.r.t. [17, 16], i.e. lower constraints are the strongest ones and the constraint composition $\otimes$ is the *glb* operator.

1. $c \vdash \exists_x c$
2. if $c \vdash d$ then $\exists_x c \vdash \exists_x d$
3. $\exists_x (c \otimes \exists_x d) \sim \exists_x c \otimes \exists_x d$
4. $\exists_x (\exists_y c) \sim \exists_y (\exists_x c)$

$\forall x, y, z \in V$, $\forall c \in C$, the diagonal element d_{xy} satisfies

1. $d_{xx} \sim true$
2. if $z \neq x, y$ then $d_{xy} \sim \exists_z (d_{xz} \otimes d_{zy})$
3. if $x \neq y$ then $d_{xy} \otimes \exists_x (c \otimes d_{xy}) \vdash c$

In the following, we denote by $\hat{x}$ both a tuple and a set of variables. For syntactic convenience, given $\hat{x} = (x_1, \ldots, x_n)$ and $\hat{y} = (y_1, \ldots, y_n)$, the notation $\exists_{\hat{x}} c$ stands for $\exists_{x_1}(\ldots \exists_{x_n}(c)\ldots)$, while $d_{\hat{x}\hat{y}}$ stands for $d_{x_1 y_1} \otimes \ldots \otimes d_{x_n y_n}$.

Example 1. Let $\Sigma = \{a/0, b/0, \ldots, f/n, g/n, \ldots\}$ be a finite set of function symbols with arity, and V be a finite set of variables. Consider the first order language defined over the term system induced by Σ, by using syntactic equality as unique predicate symbol. The Herbrand constraint system C_H has atomic propositions as elementary assertions and an entailment relation satisfying Clark's equality axioms. Cylindrification $\exists$ is the usual existential quantification, while diagonal elements are $d_{xy} \sim (x = y)$. Thus constraints are equivalent to quantified syntactic equation systems.

Example 2. Let V be a finite set of variables and ϕ be a property. The elementary assertions in the constraint system of variable dependencies $DEP_\phi(V)$ are pairs of variable's subsets. Assertion (A, B) means that if all the variables in B satisfy property ϕ, then all the variables in A satisfy the property too.
The entailment relation is defined as follows. If $A \subseteq B$ then $\emptyset \vdash (A, B)$; if $R \vdash (A, B)$ and $R \vdash (B, C)$ then $R \vdash (A, C)$; finally, if $R \vdash (A, C)$ and $R \vdash (B, D)$ then $R \vdash (A \cup B, C \cup D)$. $\exists_x R \sim \delta(R) \setminus \{(A, B) \in R \mid x \in A \cup B\}$ is the cylindrification operator, while diagonal elements are $d_{xy} \sim \{(\{x\}, \{y\}), (\{y\}, \{x\})\}$. The disjunctive completion of this constraint system is isomorphic to the constraint system *Prop*. We can easily associate with each dependency relation $R = \{(A_i, B_i) \mid 1 \leq i \leq n\}$ the propositional formula $\bigwedge_{i=1}^{n} (\wedge A_i \leftarrow \wedge B_i)$. In the following we will use the simpler *Prop* representation.

3 The Language

In this section we introduce concurrent constraint languages, as defined in [16]. The syntax and the semantics are parametric with respect to a given constraint system C. Elementary actions (**ask** and **tell**), hiding ($\exists$), parallel composition ($\|$), guarded nondeterministic choice ($\sum$) and recursion are the syntactic operators (see Table 1). For notational convenience, we write $\bigoplus_{i=1}^{n} A_i$ to denote the pure nondeterministic choice operator (*local* choice), namely the agent $\sum_{i=1}^{n} \mathbf{ask}(true) \rightarrow A_i$.

$$
\begin{array}{ll}
\text{Progr} ::= \text{Dec} \;.\; \text{Agent} & \text{Agent} ::= \textbf{tell}(c) \\
 & \qquad\quad \mid\; \exists \hat{x}.\text{Agent} \\
 & \qquad\quad \mid\; \text{Agent} \,\|\, \text{Agent} \\
\text{Dec} \quad ::= \epsilon & \qquad\quad \mid\; \sum_{i=1}^{n} (\textbf{ask}(c_i) \rightarrow \text{Agent}_i) \\
\qquad\quad \mid\; p(\hat{x}) \;:-\; \text{Agent} \;.\; \text{Dec} & \qquad\quad \mid\; p(\hat{y})
\end{array}
$$

Table 1: The syntax

3.1 Operational Semantics

The operational model is 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 relation satisfying axioms **R1**–**R5** of Table 2.

$$
\begin{array}{ll}
\textbf{R1}\ \langle\, \textbf{tell}(c)\,,\ \sigma\,\rangle \longrightarrow \langle\, \epsilon\,,\ \sigma \otimes c\,\rangle & \textbf{R4}\ \dfrac{\langle\, A\,,\ \sigma\,\rangle \longrightarrow \langle\, A'\,,\ \sigma'\,\rangle}{\begin{array}{c}\langle\, A \| B\,,\ \sigma\,\rangle \longrightarrow \langle\, A' \| B\,,\ \sigma'\,\rangle \\ \langle\, B \| A\,,\ \sigma\,\rangle \longrightarrow \langle\, B \| A'\,,\ \sigma'\,\rangle\end{array}} \\
\textbf{R2}\ \dfrac{\sigma \vdash c_i}{\langle\, \sum_{i=1}^{n} (\textbf{ask}(c_i) \rightarrow A_i)\,,\ \sigma\,\rangle \longrightarrow \langle\, A_i\,,\ \sigma\,\rangle} & \\
\textbf{R3}\ \dfrac{\langle\, A\,,\ d \otimes \exists_{\hat{x}} \sigma\,\rangle \longrightarrow \langle\, B\,,\ e\,\rangle}{\langle\, \exists(\hat{x}, d).A\,,\ \sigma\,\rangle \longrightarrow \langle\, \exists(\hat{x}, e).B\,,\ \sigma \otimes \exists_{\hat{x}} e\,\rangle} & \textbf{R5}\ \dfrac{p(\hat{x}) \;:-\; A \in P}{\langle\, p(\hat{y})\,,\ \sigma\,\rangle \longrightarrow \langle\, \Delta^{\hat{y}}_{\hat{x}}.A\,,\ \sigma\,\rangle}
\end{array}
$$

Table 2: The transition system

The execution of an elementary action $\textbf{tell}(c)$ simply adds the constraint c to the current store σ (no consistency check). A guard $g_i = \textbf{ask}(c_i)$ in the nondeterministic choice operator is a global test. It is *enabled* if the current store σ is strong enough to entail the constraint c (i.e. when $\sigma \vdash c$). The nondeterministic choice operator selects one enabled guard g_i and then behaves like the agent A_i. If no guards are enabled, then it suspends, waiting for other agents to add more information to the store. Axiom **R3** describes the hiding operator. The syntax is extended to deal with a local store d holding information about the hidden variables $\hat{x}$. Hence the information about $\hat{x}$ 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. $\exists\hat{x}.A \equiv \exists(\hat{x}, true).A$. Parallelism is modeled as *interleaving* of basic actions. Processes A and B never communicate synchronously in $A\|B$. Finally, when executing a procedure call, $\Delta^{\hat{y}}_{\hat{x}}A$ denotes the agent $\exists\hat{\psi}.(\textbf{tell}(d_{\hat{y}\hat{\psi}})\|\exists\hat{x}.(\textbf{tell}(d_{\hat{\psi}\hat{x}})\|A))$ and models parameter passing without variable renaming (variables in $\hat{x}$ can occur in $\hat{y}$). Variables $\hat{\psi}$ are special, meaning that they are not allowed to occur in user programs.

Let $\not\longrightarrow$ denote the absence of admissible transitions. Sequences of transitions reaching configurations $\langle\, A_n, c_n\,\rangle$ such that $\langle\, A_n, c_n\,\rangle \not\longrightarrow$ are called *terminating* sequences and $c_n \in B_C$ is the answer constraint. If A_n contains some guarded choice operators then the corresponding sequence is *suspended*, otherwise it is a *successful* sequence (in the latter case we denote A_n by ϵ).

Definition 2. The *finite semantics* for program $P = D.A$ is

$$\mathcal{O}_D[\![A]\!] = \lambda\sigma. \left\{ c \;\middle|\; \langle A , \sigma \rangle \xrightarrow{*} \langle B , c \rangle \not\longrightarrow \right\}$$

Note that the finite semantics observes answer constraints associated with terminating configurations, regardless of whether the associated computations are successful or suspended.

3.2 Denotational Semantics

The standard denotational semantics for concurrent constraint languages models processes as sets of reactive sequences [6] or trace operators [16]. In this paragraph we consider the simpler denotational semantics modeling the *angelic* language [14], i.e. the language obtained by replacing the global choice operator by the local choice operator. This semantics is a suitable base for reasoning about synchronization approximation, since it separates the choice operator from the synchronization operator (while in the standard semantics their interaction gives the so-called *demonic* nondeterminism or *indeterminism*).

The angelic transition system T', yelding the operational semantics $\mathcal{O}'$, is obtained by imposing $n = 1$ in rule **R2** of Table 2 and by adding rule **R6**: $\langle \bigoplus_{i=1}^{n} A_i , \sigma \rangle \longrightarrow \langle A_i , \sigma \rangle$.

[16] defines the finite semantics of *deterministic cc* languages (without choice operator) as a lower closure operator[5] (lco) on B_C (the set of finite elements of the constraint system C), mapping divergent computations to $false$. A lco on a complete lattice is characterized by its image (i.e. the set of fixpoints). Furthermore, lco's form a complete lattice. By using the fixpoint representation, we have that the pointwise ordering is $\subseteq$, the bottom element is $\{false\}$ (i.e. $\lambda x.false$), the top element is C (i.e. id) and the *glb* is given by set intersection.

Since the local choice operator introduces nondeterminism, we have to consider *sets* of constraints in order to model the computational behaviour, because in general the *lub* of two constraints is weaker than their disjunction. Intuitively, we want to record the *minimal guarantee* of a set of constraints, i.e. the pre-order: $S_1 \sqsubseteq S_2$ iff $\forall c \in S_1 \exists\, d \in S_2 \,.\, c \vdash d$.

Definition 3. Given a partial order $\langle C, \leq_C \rangle$, the downward closure of $S \subseteq C$ is defined by $down(S) = \{d \in C \mid \exists c \in S.d \leq_C c\}$. A subset S is downward closed iff $S = down(S)$. Given a function $f : C \to \wp(C')$, the downward closure of f is the function $g = Down(f) : C \to \mathcal{P}{\downarrow}(C')$ such that $g(c) = down(f(c))$.

[5] Recall that we are ordering the constraint system in a dual fashion. Lower closure operators and downward-closed sets of constraints correspond to upper closure operators and upward-closed sets of constraints in [16] and [14].

By identifying sets of constraints that are equivalent with respect to $\sqsubseteq$, we obtain a domain isomorphic to the complete lattice $\mathcal{P}{\downarrow}(C)$ of *downward-closed* subsets of C. The partial order is $\leq \equiv \subseteq$, the *lub* and the *glb* are given by set union and set intersection respectively. Furthermore the *immersion* function $\downarrow : C \to \mathcal{P}{\downarrow}(C)$ is given by ${\downarrow}c = down(\{c\})$.

Since for *cc* programs disjunction arises only when considering alternative computations, the finite semantics of angelic processes is modeled as a *linear* lco (llco) on $\mathcal{P}{\downarrow}(C)$ [14], i.e. a lco f satisfying $f(\cup S_i) = \cup f(S_i)$. A llco f is fully characterized by the set $SF(f) \subseteq C$ of its singleton fixpoints, i.e. constraints c such that $f({\downarrow}c) = {\downarrow}c$. By using this characterization we easily see that $llco(\mathcal{P}{\downarrow}(C))$ (the set of llco's on $\mathcal{P}{\downarrow}(C)$) is a complete lattice with *lub* and *glb* given by set union and set intersection respectively.

Table 3 shows the angelic semantic functions $\mathcal{E}$, $\mathcal{D}$ and $\mathcal{N}$. *Env* is the set of environments, i.e. the set of functions from process names to their denotation in $llco(\mathcal{P}{\downarrow}(C))$. Note that the denotational semantics actually extends to the *cc* paradigm the C-semantics of pure logic programs [10].

Proposition 4. $\mathcal{O}_D[\![A]\!](c) \subseteq \mathcal{O}'_D[\![A]\!](c) \subseteq Down(\mathcal{O}'_D[\![A]\!])(c) = \mathcal{N}[\![D.A]\!]({\downarrow}c)$

$$
\begin{array}{|l|}
\hline
\mathcal{E} : \mathrm{A}gent \times Env \to llco(\mathcal{P}{\downarrow}(C)) \\
\mathcal{E}[\![\mathbf{tell}(c)]\!]e = {\downarrow}c \\
\mathcal{E}[\![\mathbf{ask}(c) \to A]\!]e = \{d \in C \mid d \vdash c \;\Rightarrow\; d \in \mathcal{E}[\![A]\!]e\} \\
\mathcal{E}[\![\exists\hat{x}.A]\!]e = \{d \in C \mid \text{there exists } c \in \mathcal{E}[\![A]\!]e \text{ s.t. } \exists_{\hat{x}}c = \exists_{\hat{x}}d\} \\
\mathcal{E}[\![A \parallel B]\!]e = \mathcal{E}[\![A]\!]e \;\cap\; \mathcal{E}[\![B]\!]e \\
\mathcal{E}[\![\bigoplus_{i=1}^{n} A_i]\!]e = \bigcup_{i=1}^{n} \mathcal{E}[\![A_i]\!]e \\
\mathcal{E}[\![p(\hat{y})]\!]e = \left\{ d \in C \;\middle|\; d = \exists_{\hat{\psi}}(d_{\hat{y}\hat{\psi}} \otimes c),\, c \in (e\ p) \right\} \\
\\
\mathcal{D} : \mathrm{D}ec \times Env \to Env \\
\mathcal{D}[\![\epsilon]\!]e = e \\
\mathcal{D}[\![p(\hat{x}) \;:-\; A \,.\, D]\!]e = \mathcal{D}[\![D]\!] \left(e \left[p \mapsto \mathcal{E}[\![\exists\hat{x}.(\mathbf{tell}(d_{\hat{\psi}\hat{x}}) \parallel A)]\!]e \right] \right) \\
\\
\mathcal{N} : \mathrm{P}rogr \to llco(\mathcal{P}{\downarrow}(C)) \\
\mathcal{N}[\![D.A]\!] = \mathcal{E}[\![A]\!](lfp\, \mathcal{D}[\![D]\!]) \\
\hline
\end{array}
$$

Table 3: The finite angelic semantic operators

4 Program Properties and Approximations

The operational semantics of a *cc* program associates each initial store c to the set of all the answer 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 the constraint system, namely the set of constraints that satisfy the property ϕ. Thus, a program satisfies a semantic property ϕ iff (for each initial store) the observables of the program are a *subset* of the property, i.e. for all $c \in C \,.\, \mathcal{O}_D[\![A]\!](c) \subseteq \phi$. Following this general view, also pursued in [7] in the context of partial correctness proofs for *cc* programs, we can formalize the static analysis as a finite construction of an approximation (a superset) of the program denotation. If the approximation satisfies the semantic property, then we can safely say that our program satisfies the property too.

Let us define a program property to be *ordering closed* iff it is downward closed or upward closed. Ordering closed properties are easier to verify, as shown by the following proposition.

Proposition 5. *A program $P = D.A$ satisfies a downward closed (upward-closed) property $\phi \subseteq C$ iff the downward closure (upward closure) of $\mathcal{O}_D[\![A]\!]$ satisfies ϕ.*

Simplification arises because we can base our abstract interpretation framework on a semantics that returns ordering closed observables. An example of downward closed property is *definiteness*. If a variable x is fully instantiated in a constraint c then it is fully instantiated in all the constraints d such that $d \vdash c$.

The adjoint framework of [5] is a powerful tool for the analysis of ordering closed properties. The *best* approximations for both concrete objects and semantic functions always exist and we can compare accuracy of different abstract semantic functions. Consider downward closed properties.

Definition 6. Let $\langle M, \leq, \sqcup, \sqcap \rangle$ and $\langle M', \leq', \sqcup', \sqcap' \rangle$ be complete lattices. An *upper Galois connection* between M and M' is a pair of functions $\langle \alpha, \gamma \rangle$ such that: $\forall x \in M \,.\, \forall y \in M' \,.\, \alpha(x) \leq' y \Leftrightarrow x \leq \gamma(y)$. An upper Galois *insertion* between M and M' (denoted by $\langle M, \alpha, \gamma, M' \rangle$) is an upper Galois connection such that α is surjective (equivalently, γ is one-to-one).

Upper Galois insertions are commonly used in abstract interpretation of (constraint) logic languages. Here the approximation process returns weaker (w.r.t. $\vdash$) semantic objects (an example for the *clp* case is in [11]).

5 Generalized Abstract Interpretation

Generalized abstract interpretation is intended to perform static analysis using the *same* semantic construction for both the concrete and abstract computations. Given an abstract constraint system $\mathcal{A}$ that correctly approximates the concrete constraint system C, the program P computing on C is syntactically transformed into a program P' computing on $\mathcal{A}$. The static analysis of P is obtained by computing the semantics of P'.

Definition 7. A constraint system $\langle \mathcal{A}, \vdash', false', true', \otimes', V, \exists'_x, d'_{xy} \rangle$ is upper *correct* with respect to the constraint system $\langle C, \vdash, false, true, \otimes, V, \exists_x, d_{xy} \rangle$, using a surjective and monotonic function $\alpha : C \to \mathcal{A}$, iff (for each $c \in C$, $x, y \in V$) $\alpha(\exists_x c) \vdash' \exists'_x \alpha(c)$ and $\alpha(d_{xy}) \vdash' d'_{xy}$.

Proposition 8. *If $\mathcal{A}$ is upper correct w.r.t. constraint system C using α then there exists an upper Galois insertion relating $\mathcal{P}{\downarrow}(C)$ and $\mathcal{P}{\downarrow}(\mathcal{A})$.*

Example 3. Let C_H be the Herbrand constraint system, let DEP_g be the dependency relation between variables induced by groundness and let the function sol map an equational constraint into its (equivalent) solved form. Define $\alpha_g : C_H \to DEP_g$ as follows.

$$\alpha_g(c) = \begin{cases} \exists_{\tilde{y}} \left(\cup \left\{ \{(\{x_i\}, var(t_i)), (var(t_i), \{x_i\})\} \;\middle|\; x_i = t_i \in E \right\} \right) & \text{if } sol(c) = \exists_{\tilde{y}} E \\ False & \text{if } sol(c) = false \end{cases}$$

Proposition 9. DEP_g *is upper correct w.r.t. C_H using α_g.*

As usual, termination is guaranteed by assuming finite constraint systems.

5.1 The Abstract Synchronization Problem

Consider the angelic *cc* language and let f be a llco on $\mathcal{P}{\downarrow}(C)$ (the concrete semantics of an agent). Let $\mathcal{A}$ be an abstract constraint system upper correct w.r.t. C using α and let $(\tilde{\alpha}, \gamma)$ be the induced upper Galois insertion relating the concrete domain $\mathcal{P}{\downarrow}(C)$ and the abstract domain $\mathcal{P}{\downarrow}(\mathcal{A})$. The *best correct approximation* for f on $\mathcal{P}{\downarrow}(\mathcal{A})$ is $f^\sharp = (\tilde{\alpha} \circ f \circ \gamma)$. Let $f' : \mathcal{P}{\downarrow}(\mathcal{A}) \to \mathcal{P}{\downarrow}(\mathcal{A})$ be an abstract semantic operator. f' is a *correct* upper approximation of f on $\mathcal{P}{\downarrow}(\mathcal{A})$ iff $f^\sharp \vdash' f'$ [5].

Proposition 10. $f^\sharp = (\tilde{\alpha} \circ f \circ \gamma)$ *is a llco on $\mathcal{P}{\downarrow}(\mathcal{A})$.*

The abstract and the concrete semantics of angelic processes can be modeled in the same way and we can write $f^\sharp \vdash' f'$ as $f^\sharp \subseteq f'$. However, the simple transformation considered in [11] is no longer admissible for *cc* programs because the abstract synchronization operator would not be correct. The following theorem justifies this observation.

Theorem 11.

$$\begin{bmatrix} \forall\, c \in C,\ \forall\, f \in llco(\mathcal{P}{\downarrow}(C)),\ f' \in llco(\mathcal{P}{\downarrow}(\mathcal{A})) \\ \text{s.t. } f^\sharp \subseteq f' \\ [\mathbf{ask}(c) \to f]^\sharp \subseteq \mathbf{ask}(\alpha(c)) \to f' \end{bmatrix} \Leftrightarrow \left[\alpha \text{ is an isomorphism} \right]$$

A "solution" to the abstract synchronization problem can be found by considering a different (more abstract) concrete semantics which models only some aspects of the program behaviour.

Definition 12. The *success semantics* for program $P = D.A$ is

$$\mathcal{SS}_D[\![A]\!] = \lambda\, \sigma \in C \,.\, \left\{ c \;\middle|\; \langle A, \sigma \rangle \stackrel{*}{\longrightarrow} \langle \epsilon, c \rangle \right\}$$

This semantics does not observe answer constraints associated to suspended computations. It observes successful computations only.

Proposition 13. *If $P = D.A$ is suspension-free then* $\mathcal{O}_D[\![A]\!] = \mathcal{SS}_D[\![A]\!]$.

Turning our attention to the success semantics, we easily see that correctness depends on the following condition:

"concrete computation proceeds $\Rightarrow$ abstract computation proceeds"

Thus, whenever we cannot prove the contrary, we assume that the concrete computation proceeds. The simplest way to satisfy this correctness condition consists in removing all synchronizations from the program. Consider the transformation *NoSynch* : Progr $\rightarrow$ Progr, acting on nondeterministic choice operators only.

$$\mathit{NoSynch}[\sum_{i=1}^{n} (\mathbf{ask}(c_i) \rightarrow A_i)] = \bigoplus_{i=1}^{n} (\mathbf{tell}(c_i) \parallel \mathit{NoSynch}[A_i])$$

In the following let $P_1 = D_1.A_1 = \mathit{NoSynch}[P]$. Since we have discarded every meaningful synchronization test, processes in P_1 always proceed, providing a correct approximation of the success semantics of P.

Proposition 14. $\mathcal{SS}_D[\![A]\!](c) \subseteq \mathcal{SS}_{D_1}[\![A_1]\!](c)$

Transformed programs are very similar to sequential constraint logic programs. As in [11], their semantics can easily be modeled by a single (disjunctive) constraint (see [12]). For space reasons, we omit the presentation of this simpler semantics and use Table 3 again, where the equation dealing with asks has become useless.

Corollary 15. $\mathit{Down}(\mathcal{SS}_{D_1}[\![A_1]\!]) = \mathcal{N}[\![D_1.A_1]\!]$

To obtain the abstract semantics of the transformed program P_1 we now apply the generalized approach of [11]. This amounts to instantiate Table 3 over the abstract constraint system $\mathcal{A}$ and then to compute the semantics of the abstract program P_1' corresponding to P_1.

Definition 16. Given the program P on C, the *corresponding abstract program* P' on $\mathcal{A} = \alpha(C)$ is obtained by replacing each *tell* constraint c of P by $\alpha(c)$.

Note that this transformation does not affect ask constraints. Let P_1' be the abstract program corresponding to P_1.

Theorem 17. $(\tilde{\alpha} \circ \mathcal{N}[\![P]\!] \circ \gamma) \subseteq \mathcal{N}'[\![P']\!]$

Example 4. Consider the program P which appends two lists.

```
app(X,Y,Z) :- ask(X=[]) -> tell(Y=Z)
            + ask(∃H,X1 X=[H|X1]) ->
```

```
                ∃ H,X1,Z1. tell(X=[H|X1],Z=[H|Z1]) || app(X1,Y,Z1).
```

$P_1 = \mathit{NoSynch}[P]$, after a straightforward simplification, is

```
app(X,Y,Z) :- tell(X=[],Y=Z)
           ⊕ ∃ H,X1,Z1. tell(X=[H|X1],Z=[H|Z1]) || app(X1,Y,Z1).
```

Let us consider the abstract constraint system $\mathcal{A} = \mathit{Prop}$. The program P_1' on *Prop* corresponding to P_1 is

```
app(X,Y,Z) :- tell(X∧Y↔Z)
           ⊕ ∃ H,X1,Z1. tell(X↔(H∧X1)∧Z ↔(H∧Z1)) || app(X1,Y,Z1).
```

We obtain $\mathcal{N}'[\![P_1'.app(x,y,z)]\!](\downarrow a) = \downarrow a \cap \downarrow((x \wedge y) \leftrightarrow z)$ specifying that in all the answer constraints associated to *successful* computations of the original program, the third argument of *app* is bound to a ground term iff both the first and the second argument are bound to ground terms.

6 An "Angelic" Solution

To approximate the *standard* semantics of a program without any suspension freeness information, e.g. if we are trying to prove suspension-freeness, the previous approach is no longer applicable. As an alternative, we can consider the best correct lower approximation of the synchronization operator, that is

$$[\mathbf{ask}(c) \to f]^\sharp = \lambda S' \in \mathcal{P}{\downarrow}(\mathcal{A}).\cup\{\mathbf{if}\ \gamma(\downarrow a) \subseteq (\downarrow c)\ \mathbf{then}\ f^\sharp(\downarrow a)\ \mathbf{else}\ \downarrow a \mid a \in S'\ \}$$

Note however that in general we cannot statically compute the concrete object $\gamma(a)$. In practice, we have to implement a "hybrid" synchronization test which verifies whether an abstract constraint *definitely entails* a concrete one,

$$\mathit{test} : (\mathcal{A} \times C) \to \mathit{Bool} \quad \text{such that} \quad \mathit{test}(a,c) = \mathit{true} \;\Rightarrow\; \gamma(\downarrow a) \subseteq (\downarrow c).$$

Informally, this condition means "if the abstract computation proceeds then every concrete computation it approximates proceeds too".
Notice that this approach is not based on generalizing the semantics, as the abstract program does not perform all computations on the abstract constraint system. Moreover, this synchronization primitive strongly depends upon the specific analysis, i.e. upon the specific choice of the abstract domain.

Given a synchronization primitive, we have to choose a suitable approximation of the nondeterministic operator. In order to get an efficient abstract interpretation framework, we cannot directly abstract global choice, since the associated denotational models are too complex [2, 8]. Local choice (i.e. angelic languages) seems to be a good cost/precision tradeoff.

Consider the transformation *Angel*, mapping don't care choice operators into don't know choice operators with *multiple synchronization*.

$$\mathit{Angel}[\sum_{i=1}^{n}(\mathbf{ask}(c_i) \to A_i)] = \mathbf{ask}(c_1;\ldots;c_n) \to \bigoplus_{i=1}^{n} (\mathbf{tell}(c_i) \parallel \mathit{Angel}[A_i])$$

The meaning of a multiple synchronization test is to ask the *disjunction* of all the guard constraints: $\forall \sigma \in S \,.\, \exists j \in \{1, \ldots, n\} \,.\, (\downarrow\sigma) \subseteq (\downarrow c_j) \;\Leftrightarrow\; S \subseteq \bigcup_{i=1}^{n} (\downarrow c_i)$.

Remark. This is not true when we consider a *widening* as disjunction operator. As an example, consider a constraint system dealing with rational intervals with entailment given by inclusion. Process $\mathbf{ask}(x \in [0,1]; x \in [1,2]) \rightarrow A$ and its widened version $\mathbf{ask}(x \in [0,2]) \rightarrow A$ are not equivalent. Given the initial store $x \in [0,2]$, the first computation (correctly) suspends, while the latter proceeds, possibly providing uncorrect results.

The denotational semantics of this kind of programs can be obtained by using Table 3 and by replacing the equation for the (simple) synchronization operator with the following equation.

$$\mathcal{E}[\![\, \mathbf{ask}(c_1; \ldots; c_n) \rightarrow A \,]\!]e = \{d \in C \mid \exists\, i \in \{1, \ldots, n\}. d \vdash c_i \quad \Rightarrow \quad d \in \mathcal{E}[\![\, A \,]\!]e\,\}$$

Therefore, given $S' \in \mathcal{P}\!\downarrow\!(\mathcal{A})$ and $c_1, \ldots, c_n \in C$, the *multiple abstract synchronization test* has to satisfy

$$mtest(S', c_1; \ldots; c_n) = true \quad \Rightarrow \quad \forall \sigma \in \gamma(S') \,.\, \exists i \in \{1, \ldots, n\} \,.\, \sigma \vdash c_i$$

The abstract semantics of a transformed program is computed by using Table 3 (instantiated over $\mathcal{A}$) and by replacing the equation dealing with simple ask by the following

$$\mathcal{E}'[\![\mathbf{ask}(c_1; \ldots; c_n) \rightarrow A\,]\!]e = \left\{ a \in \mathcal{A} \;\middle|\; mtest(\downarrow\! a, c_1; \ldots; c_n) = true \;\Rightarrow\; a \in \mathcal{E}'[\![\, A \,]\!]e \right\}$$

Example 5. The producer **pzaff** sends messages to consumers **cgiaco** and **clevi** by using a single channel. For each input message, the distributor **distr** forwards the text to the appropriate output channel.

```
pzaff(X) :-
   ask(true) -> ∃ Y,M. tell(X=[msg(levi,M)|Y]) || write(M) || pzaff(Y)
      +
   ask(true) -> ∃ Y,M. tell(X=[msg(giaco,M)|Y]) || write(M) || pzaff(Y)
      +
   ask(true) -> tell(X=[]).

distr(X,L,G) :-
   ask(∃T,X1 X=[msg(levi,T)|X1]) ->
      ∃ T,X1,L1. tell(X=[msg(levi,T)|X1],L=[T|L1]) || distr(X1,L1,G)
      +
   ask(∃T,X1 X=[msg(giaco,T)|X1]) ->
      ∃ T,X1,G1. tell(X=[msg(giaco,T)|X1],G=[T|G1]) || distr(X1,L,G1)
      +
   ask(X=[]) -> tell(L=[],G=[]).

g(X,L,G) :- pzaff(X) || distr(X,L,G) || clevi(L) || cgiaco(G).
```

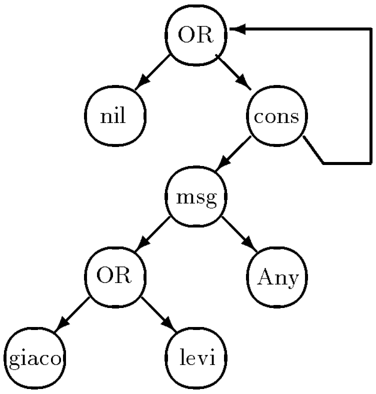

Fig. 1. The rigid typegraph for X

Assuming that `write`, `clevi` and `cgiaco` are suspension free, the suspension freeness of `g(X,L,G)` may only depend on `pzaff` and `distr`. By applying the *Angel* transformation, we note that the only process that can suspend is `distr`. Suspension freeness can be analyzed by evaluating the following multiple ask

$$(\exists_{\mathtt{T},\mathtt{X1}}\mathtt{X=[msg(levi,T)|X1]})\ ;\ \exists_{\mathtt{T},\mathtt{X1}}\mathtt{X=[msg(giaco,T)|X1]})\ ;\ \mathtt{X=[]})$$

For this purpose, the *rigid types* abstraction [15] provides an adequate abstract domain. Intuitively, the process `pzaff` binds the variable `X` to any of the terms described by the rigid type graph in Figure 1. Therefore, we have to show that all such terms satisfy the synchronization test. In this case it is an easy task. However, in other cases it is necessary to extend the abstract domain of rigid types with some kind of variable dependency information (we are currently working on a formal solution for the general case).

Given $P = D.A$, let P'_1, P'_2 be the abstract programs corresponding to $P_1 = \mathit{NoSynch}[P]$ and $P_2 = \mathit{Angel}[P]$ respectively.
Next proposition states *Angel* correctness wrt the standard semantics, provided that we defined a correct multiple abstract synchronization test.

Proposition 18. *$\mathcal{N}[\![P]\!] \subseteq \mathcal{N}[\![P_2]\!]$ and $(\tilde{\alpha} \circ \mathcal{N}[\![P]\!] \circ \gamma) \subseteq \mathcal{N}'[\![P'_2]\!]$.*

If P is suspension-free, we can compare the accuracy of the two transformations, showing that *NoSynch* is always better than *Angel*.

Proposition 19. *$\mathcal{N}[\![P_1]\!] \subseteq \mathcal{N}[\![P_2]\!]$ and $\mathcal{N}'[\![P'_1]\!] \subseteq \mathcal{N}'[\![P'_2]\!]$.*

The two transformations have the same precision degree only when *Angel* can actually "prove" suspension-freeness.

7 Related Works

The initial approaches to the static analysis of concurrent logic languages are based on the operational semantics [4, 2]. In particular, in order to get independence from the scheduling policy, [2] analyze *cc* programs (with consistency check) by using a non-standard (operational) semantics that makes the computation confluent. Our program transformation *Angel* can be seen as the denotational translation of this approach (modulo the absence of the consistency check). To our knowledge, [8] defines the first abstract interpretation framework for *cc* programs based on a denotational (and compositional) semantics. Even in this case there is a two level approximation. The standard semantics is first abstracted by considering a semantics recording the input/output relation between concrete constraints, and then the constraint system is abstracted, by assuming the existence of a correct abstract synchronization test. Global choice operators are simply mapped into local choice operators. This is a heavy approximation, because one blocked guard causes the suspension of the process, even if there are other definitely enabled guards in the choice operator (e.g. in a deterministic choice we always suspend, because there can be only one enabled guard at a time).

[3] considers the problem of giving a generalized abstract interpretation framework for *cc* languages, where only local choice is allowed. In contrast with Theorem 11, they claim that it is correct to directly abstract the program and evaluating it on the abstract constraint system. However, simple counterexamples show that in general this approach returns uncorrect results because of the abstract synchronization problem.

A more recent paper [9] considers the analysis of compositionally confluent *cc* programs and defines an abstract interpretation framework which is very similar to that obtained by our transformation *Angel*. This approach, based on the denotational semantics of angelic *cc*, maps each (non-confluent) guarded choice operator into the agent $\mathbf{ask}(\bigvee_{i=1}^{n} c_i) \rightarrow \bigoplus_{i=1}^{n} A_i$, where $\vee$ denotes the disjunction over $\mathcal{P}{\downarrow}(C)$. The difference is that, once the synchronization test is passed, this transformation does not use the guard constraints to strength each branch of the computation. On the contrary, *Angel* tells each branch's guard before proceeding in the abstract computation, obtaining better results.

8 Conclusions

We have shown that the ask operators cannot be safely *upper* approximated using the generalized semantics approach. The interest in a solution to this problem in the context of abstract interpretation is not only related to the analysis of *cc* programs. Indeed, the basic problem in the abstraction of synchronization for *cc* programs is shared by a number of different semantic constructions, not necessarily related with the ask-based synchronization of concurrent languages. As shown in [1], the semantics of (pure) Prolog programs (logic programs with depth-first search) can be specified in term of *implicit ask mappings*. A reduction with a clause can only be applied to a goal provided that there are no infinite branches on the left-hand side of the proof tree for that goal, by applying any of the previous clauses in the textual order. A similar behaviour is also shared by semantic models for builtins in Prolog. While the *implicit ask mapping*-based semantic definitions for Prolog's search or builtins provide a more declarative model for control features in standard Prolog interpreters, their use as semantic bases for abstract interpretation may lead to some of the problems discussed in the previous sections. It is interesting to note that, in the case of Prolog depth-first search, a *NoSynch*-like abstraction approximates the program meaning (the Prolog success set) by its interpretation as a pure logic program (i.e. without depth-first search). This, indeed, is a common practice in data-flow analysis of Prolog programs (a discussion on this topic is in [1]).

References

1. R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 95–104. ACM Press, 1992.
2. 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.
3. 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.
4. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 215–232. The MIT Press, Cambridge, Mass., 1990.
5. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
6. 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.
7. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 98–108. ACM Press, 1994.
8. 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.

9. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. Technical report, Dipartimento di Elettronica e Informatica, University of Padova, 1993.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
11. 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.
12. R. Giacobazzi, G. Levi, and E. Zaffanella. Abstracting Synchronization in Concurrent Constraint Programming. Technical Report TR 25/93, Dip. di Informatica, Univ. di Pisa, 1993.
13. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.
14. R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Science Lab., Xerox PARC, 1991.
15. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
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.

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