The Journal of Functional and Logic Programming

The MIT Press

Volume 1997, Article 6

5 November, 1997

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

The Journal of Functional and Logic Programming is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

Abstracting Synchronization in Concurrent Constraint Programming

Enea Zaffanella Roberto Giacobazzi Giorgio Levi

5 November, 1997

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some of the most important techniques for data-flow analysis of (sequential) constraint logic programs (*clp*) are no longer applicable to *cc* languages.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 that may be influenced by abstraction) becomes useless for relevant applications. In the case of *clp* programs, abstract interpretation of a program P is obtained by evaluating an abstract program $\alpha(P)$ into an instance of *clp* itself, provided with a suitable abstract constraint system. In *cc* programs, a correct characterization of suspended computations can only be obtained by replacing **ask** constraints with stronger constraints, which is not the case in abstract interpretation, where abstraction is usually a weakening of constraints. A possible solution to this problem is based on a more abstract (nonstandard) semantics: the success semantics, which models nonsuspended computations only. With a program transformation (NoSynch) that simply ignores synchronization, we obtain a *clp*-like program that 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. The resulting abstraction 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 nonstandard domains. The standard domain of values is replaced by a domain of descriptions of values, and the basic operators are provided with a corresponding nonstandard interpretation. In the classical framework of abstract interpretation [CC77], 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 $\gamma(\mathcal{I}(P))$. This evaluation should provide a finite (and therefore approximated) description of the program behavior, so as to determine (at compile time) run-time properties of the program. The approach is general enough to be domain independent and language independent: by formalizing a domain abstraction, it can be applied to any semantic definition, independently of the underlying programming language.

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. For instance, in logic programming, abstract interpretation is obtained by abstracting *unfolding*, which is basically *replacement* + *unification*. This corresponds precisely to defining a notion of *abstract unfolding*, which usually implements abstract unification, but leaves replacement unchanged. In this direction, the generalized approach to the semantics in [GDL95] has been introduced precisely to factorize the semantics with respect to its domain-dependent features (i.e., operators and semantic objects). This makes the above distinction between replacement and unfolding more apparent. 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 (in the sense of abstract interpretation) by abstractly evaluating an abstract version of the program. The *ask-tell* paradigm [Mah87], which is the basis of *cc* languages [SR90], 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 in a shared *store* [SRP91]. A store is a constraint representing the global state of the computation. Synchronization is achieved through *blocking ask*: the process is suspended when the store does not entail the **ask** constraint, and it remains suspended until the store entails it. This mechanism introduces some problems when dealing with abstraction. Intuitively, a correct approximation of the program meaning generates weaker answers for any possible program behavior. Thus, 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. This "negative" result, however, can be the basis for reasoning about new correct abstractions for *cc* programs. A simple solution can be obtained by considering a more abstract semantics modeling nonsuspended computations only. A transformation that 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 success semantics are equivalent. This justifies a possible preventive use of a suspension-analysis phase [CFM94, CFMW93] before generating any sound approximation of the concrete semantics of *ccp* agents.

A different approach to solving the above problem can be obtained by introducing hybrid primitives to deal with **ask** constraints. As before, we use a program transformation (*Angel*) that essentially replaces *don't care* nondeterminism with *don't know* nondeterminism. Following the semantic characterization of angelic *cc* processes given in [JSS91], we obtain the denotational counterpart of the transition-system-based suspension analysis in [CFMW93] (modulo the absence of the consistency check). Simple results relate the accuracy of these different solutions when the program is suspension-free (i.e., when the success semantics and the standard semantics are the same), showing that the first approach always gives better analysis than the second one. Moreover, while the second solution is applicable to possibly non-suspension-free programs, it is usually more complex in the semantics construction, and may require more-complex abstract domains to detect suspension freeness.

The paper is structured as follows. After preliminary definitions in Section 2, in Section 3 we introduce the notion of a constraint system for *cc* programs. In Section 4 we introduce the syntax and the operational semantics of *cc* programs. Following [SRP91], we also provide a denotational semantics for the subclass of angelic programs. In Section 5 we introduce the notion of observable program properties for processes. The abstract synchronization problem is considered in Section 6 by using the generalized approach to the semantics. We show an abstract interpretation scheme that is correct with regard to the success semantics of a *cc* program. In Section 7 we introduce an alternative solution for synchronization abstraction.

2 Preliminaries

Throughout the paper we will assume familiarity with the basic notions of lattice theory (cf. [Bir67]) and abstract interpretation (cf. [CC77, CC79b]).

Given the sets A and B, $A \backslash B$ denotes the set A where the elements in B have been removed. The powerset of a set S is denoted by $\wp(S)$. The class of finite (possibly empty) subsets of a set S is denoted $\wp f(S)$. Let Σ be a possibly infinite set of symbols. The set of objects a_i indexed on a set of symbols Σ is denoted by $\{a_i\}_{i \in \Sigma}$. The set of n tuples of symbols in Σ is denoted by Σ^n. Sequences of objects in Σ are denoted by Σ^*. Sequence length and set cardinality are both denoted by $|\ |$. Let R be a binary transitive relation on a set A, then the transitive closure of R is denoted by R^*. Syntactic identity is denoted by $\equiv$. An *algebraic structure* [HMT71] is a pair $\langle \mathcal{C}, \mathcal{Q} \rangle$ where $\mathcal{C}$ is a nonempty set, called the *universe* of the structure, and $\mathcal{Q}$ is a function ranging over an index set $\mathcal{I}$, such that for each $i \in \mathcal{I}$, $\mathcal{Q}_i$ are finitary operations or relations on $\mathcal{C}$. Algebraic structures are also denoted as $\langle \mathcal{C}, \mathcal{Q}_i \rangle_{i \in \mathcal{I}}$.

A set P equipped with a partial order $\leq$ is said to be *partially ordered*, and it is denoted $\langle P, \leq \rangle$. Given a partially ordered set $\langle P, \leq \rangle$ and $X \subseteq P$, $y \in P$ is an *upper bound* for X if and only if $x \leq y$ for each $x \in X$. An upper bound y for X is the *least upper bound* (denoted *lub*) if and only if for every upper bound y': $y \leq y'$, *lower bounds* and *greatest lower bounds* (denoted *glb*) are defined dually. A *directed set* is a partially ordered set in which any two elements, and hence any finite subset, have an upper bound in the set. A *complete lattice* is a partially ordered set L such that every subset of L has a least upper bound and a greatest lower bound. A complete lattice L with partial ordering $\leq$, least upper bound $\vee$, greatest lower bound $\wedge$, least element $\bot = \vee\emptyset = \wedge L$, and greatest element $\top = \wedge\emptyset = \vee L$, is denoted as an algebraic structure $\langle L, \leq, \bot, \top, \vee, \wedge\rangle$. In the following, we omit $\bot$, $\top$, $\vee$, and $\wedge$ when these are implicit in the definitions, and occasionally use the partially ordered set notation to denote complete lattices. Let $\langle L, \leq\rangle$ be a lattice where $x \in L$ is *finite* (in L) if for every directed set D in L: $x \leq \vee D \Rightarrow x \leq d$ for some $d \in D$. If for every $S \subseteq L$: $x \leq \vee S \Rightarrow x \leq \vee T$ for $T \in \wp^f(S)$, x is *compact*. Notice that finite and compact elements are the same in complete lattices. A complete lattice $\langle L, \leq\rangle$ is *algebraic* if for every $x \in L$: $x = \vee\{d \mid d \text{ is finite and } d \leq x\}$. An algebraic lattice is ω-algebraic if the set of its finite elements is denumerable.

We write $f : A \to B$ to mean that f is a total function of A into B. To specify function parameters in function definitions, we will often make use of Church's lambda notation. Let $f : A \to B$, for each $C \subseteq A$ we denote by $f(C)$ the image of C by f: $\{f(x) \mid x \in C\}$. Functions from a set to the same set are usually called *operators*. The identity operator $\lambda x.x$ is often 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 $f(x) \leq_B f(x')$. If and only if for each nonempty chain $X \subseteq A$: $f(\bigsqcup_A X) = \bigsqcup_B f(X)$, f is *continuous*. A function f is *additive* if and only if the previous conditions are satisfied for each nonempty 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 and idempotent operator. 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 [CC79a, Mor60]. Let $\langle L, \leq, \bot, \top, \vee, \wedge\rangle$ be a nonempty complete lattice, and $f : L \to L$. The *upper ordinal powers* of f are defined as follows:

$$
\begin{aligned}
&f \uparrow 0(X) = X \\
&f \uparrow \alpha(X) = f(f \uparrow (\alpha - 1)(X)) && \text{for every successor ordinal } \alpha\text{; and} \\
&f \uparrow \alpha(X) = \bigvee_{\delta<\alpha} f \uparrow \delta(X) && \text{for every limit ordinal } \alpha
\end{aligned}
$$

The first limit ordinal equipotent with the set of natural numbers is denoted by ω. We will denote by ω also the set of natural numbers. If f is a continuous function on a lattice, the least fixpoint $lfp(f)$ is $f \uparrow \omega(\bot)$.

3 Constraint Systems

Different formalizations of constraint systems are present in the literature [JL87, SRP91, GDL95], depending on the properties the resulting algebra has to satisfy.

The algebraic specification (for sequential constraint logic programs) given in [GDL95] 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 noncommutative and nonidempotent constraint composition operators and a wide range of (possibly nondistributive) constraint disjunction operators, i.e., widenings. On the other hand, the denotational semantics construction in [SRP91] 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 behaviors 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, which is more adequate to describe the basic notions of *consistency* and *entailment*.

The construction in [SRP91] is an extension of Scott's partial information systems [Sco82]. 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 $\vdash \subseteq \wp_f(D) \times D$. The relation $\vdash$ is a pre-order. By taking the entailment closure[1] $\delta(u)$ of a set of assertions u, we obtain the equivalence relation $\sim$ ($u \sim v$ if and only if $\delta(u) = \delta(v)$). Hence, a *simple constraint system* is $C = \langle \wp(D), \dashv \rangle /_{\sim}$, which is a complete ω-algebraic lattice [Sco82]. An

[1] 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.

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 (denumerable) *base* B of the constraint system. Bases of a constraint system C are usually denoted by B_C. To treat the hiding operator of the language, [SRP91] introduces a family of unary operations called *cylindrifications* [HMT71]. 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* [HMT71] 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 [SRP91].

Definition 1 *A (cylindric) constraint system*[2] $\langle C, \vdash, false, true, \otimes, V, \exists_x, d_{xy}\rangle$ *is an algebraic structure where:*

- $\langle C, \dashv\rangle$ *is a simple constraint system,*
- $true = [1]_\sim$ *and* $false = [0]_\sim$,
- $\otimes$ *is the glb,*
- V *is a denumerable set of variables,*
- $\forall x, y \in V$, $\forall c, d \in C$, *the operator* $\exists_x : C \to C$ *satisfies:*
  1. $c \vdash \exists_x c$,
  2. *if* $c \vdash d$ *then* $\exists_x c \vdash \exists_x d$,
  3. $\exists_x(c \otimes \exists_x d) = \exists_x c \otimes \exists_x d$, *and*
  4. $\exists_x(\exists_y c) = \exists_y(\exists_x c)$,
- $\forall x, y, z \in V$, $\forall c \in C$, *the diagonal element* d_{xy} *satisfies:*
  1. $d_{xx} = true$,
  2. *if* $z \neq x, y$ *then* $d_{xy} = \exists_z(d_{xz} \otimes d_{zy})$, *and*
  3. *if* $x \neq y$ *then* $d_{xy} \otimes \exists_x(c \otimes d_{xy}) \vdash c$.

[2] To have a standard approach when dealing with abstract interpretation, we order constraints in a dual fashion with regard to [Sco82, SRP91], i.e., lower constraints are the strongest ones, and the constraint composition $\otimes$ is the *glb* operator.

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

Example 1 (Herbrand Constraint System) *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 equality as a unique predicate symbol. The 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 $d_{xy} \equiv (x = y)$. Thus constraints are equivalent to quantified equation systems.*

The next example defines the constraint system of dependency relations between variables, and can be used for the detection of many properties (e.g., definiteness).

Example 2 (Dependency Relations [CFM94]) *Let V be a finite set of variables and p be a property. The elementary assertions are tuples of sets of variables, i.e., $(A, B) \in \mathcal{P}(V) \times \mathcal{P}(V)$. Their interpretation is the following. If all the variables in B satisfy property p, then all the variables in A satisfy the property too, i.e., $p(B) \Rightarrow p(A)$.*

The entailment relation is defined accordingly:

- *if $A \subseteq B$, then $\emptyset \vdash (A, B)$,*
- *if $R \vdash (A, B)$ and $R \vdash (B, C)$, then $R \vdash (A, C)$, and*
- *if $R \vdash (A, C)$ and $R \vdash (B, D)$, then $R \vdash (A \cup B, C \cup D)$.*

Cylindrification is defined as $\exists_x R = \delta(R) \setminus \{(A, B) \in \delta(R) \mid x \in A \cup B\}$, while diagonal elements are $d_{xy} \equiv \{(\{x\}, \{y\}), (\{y\}, \{x\})\}$.

The disjunctive completion of this constraint system is isomorphic to the constraint system Prop [GDL95]. We can easily associate the propositional formula $\bigwedge_{i=1}^{n} (\wedge A_i \leftarrow \wedge B_i)$ to the dependency relation $R = \{(A_i, B_i) \mid 1 \leq i \leq n\}$. In the following, we will use the simpler Prop representation.

4 The Language

In this section we introduce concurrent constraint languages, as defined in [SRP91]. The syntax and semantics are parametric with respect to a given constraint system C.

4.1 Syntax

The semantic operators of concurrent constraint languages are: elementary actions (**ask** and **tell**), hiding ($\exists$), parallel composition ($\|$), guarded nondeterministic choice ($\sum$), and recursion (see Table 1).

In the syntax defined in [SRP91], a process-definition body can contain free variables not occurring in the head. These are a kind of "invisible" global variables. Their presence makes the program variable-renaming dependent, and can be a source of many programming errors. In the following, we only consider *variable-renaming independent* programs.

Definition 2 (Variable-Renaming Independent Program) *Let $FV(t)$ be the set of free variables occurring in the syntactic object t. A* cc *program P is variable-renaming independent if for each process definition $p(x_1, \ldots, x_n)$:− $A \in P$, we have $FV(A) \subseteq \{x_1, \ldots, x_n\}$.*

For notational convenience, we write $\bigoplus_{i=1}^{n} A_i$ to denote the pure nondeterministic choice operator (*local* choice), namely the agent

$$\sum_{i=1}^{n} \mathbf{ask}(true) \rightarrow A_i$$

4.2 Operational Semantics

The operational model is described by a transition system $T = (Conf, \longrightarrow_T)$. Elements of *Conf* (configurations) consist of an agent and a constraint, representing the residual computation and the global store, respectively. The minimal relation satisfying axioms R1 − R5 of Table 2 is $\longrightarrow_T$.

The execution of an elementary action $\mathbf{tell}(c)$ simply adds the constraint c to the current store σ (no consistency check). A guard $g_i = \mathbf{ask}(c_i)$ in the nondeterministic choice operator is a global test. It is *enabled* if the

Program	::=	Dec . Agent
Dec	::=	ϵ
	$\mid$	$p(\vec{x})$:− Agent . Dec
Agent	::=	$\mathbf{tell}(c)$
	$\mid$	$\exists\vec{x}$.Agent
	$\mid$	Agent $\|$ Agent
	$\mid$	$\sum_{i=1}^{n}$ $(\mathbf{ask}(c_i) \rightarrow \text{Agent}_i)$
	$\mid$	$p(\vec{y})$

Table 1: The syntax

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. Parallelism is modeled as *interleaving* of basic actions. Processes A and B never communicate synchronously in $A\|B$. Axiom R4 describes the hiding operator. The syntax is extended to deal with a local store d holding information about the hidden variables $\vec{x}$. Hence the information about $\vec{x}$ produced by the external environment does not affect the process behavior and conversely the external environment cannot access the local store. Initially the local store is empty, i.e., $\exists\vec{x}.A \equiv \exists(\vec{x}, \mathit{true}).A$. Finally, when executing a procedure call, $\Delta_{\vec{x}}^{\vec{y}}A$ denotes the agent $\exists\vec{\psi}.(\mathbf{tell}(d_{\vec{y}\vec{\psi}})\|\exists\vec{x}.(\mathbf{tell}(d_{\vec{\psi}\vec{x}})\|A))$ and models parameter passing without variable renaming (variables in $\vec{x}$ can occur in $\vec{y}$). Variables $\vec{\psi}$ are special, meaning that they are not allowed to occur in user programs.

A σ-sequence s for a program $D.A$ is a possibly infinite sequence of configurations $\langle A_i, c_i \rangle_i$ such that $A_0 = A$ and $c_0 = \sigma$ and for all $i < |s|$ there exists a transition $\langle A_i, c_i \rangle \longrightarrow_T \langle A_{i+1}, c_{i+1} \rangle$. Let $\not\longrightarrow_T$ denote the absence of admissible transitions. Sequences reaching configurations $\langle A_n, c_n \rangle$ such that $\langle A_n, c_n \rangle \not\longrightarrow_T$ are called *terminating* sequences, and $c_n \in B$ is the (finite)

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

Table 2: The transition system T

answer constraint. If A_n contains some suspended choice operators, then the corresponding sequence is *suspended*; otherwise, it is a *successful* sequence, and in this case we denote A_n by ϵ.

Definition 3 *The finite semantics for program $P = D.A$ is given by the function:*

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

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

4.3 Denotational Semantics

The standard denotational semantics for concurrent constraint languages models processes as sets of reactive sequences [dBP91] or trace operators [SRP91]. In this paragraph, we consider the simpler denotational semantics modeling the *angelic* language [JSS91], 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.

In [SRP91], the finite semantics of *deterministic cc* languages (without choice operators) is defined as a lower closure operator[3] on B_C (the set of finite elements of the constraint system C), mapping divergent computations to $false$. A lower closure operator on a complete lattice is characterized by its image (i.e., the set of fixpoints). Furthermore, *lco*s form a complete lattice [War42]. 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 behavior, 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: $S_1 \sqsubseteq S_2$ if and only if $\forall c \in S_1 \exists\, d \in S_2 \,.\, c \vdash d$.

Definition 4 *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 if and only if $S = down(S)$. 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 $g(c) = down(f(c))$. Upward closures $up(S)$ and $Up(f)$ are defined dually.*

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 $\leq\, \equiv\, \subseteq$, and 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 lower closure operator on $\mathcal{P}{\downarrow}(C)$ [JSS91], i.e., a lower closure operator f satisfying $f(\cup S_i) = \cup f(S_i)$. A linear *lco* 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 linear *lco*s on $\mathcal{P}{\downarrow}(C)$) is a complete lattice with *lub* and *glb* given by set union and set intersection, respectively.

[3] 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 [SRP91] and [JSS91].

Table 3 shows the angelic semantic functions $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{N}$, which are monotonic and continuous with respect to their process arguments (*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 [FLMP89], recording the minimal guarantee of a process.

The angelic transition system T' is obtained by imposing $n = 1$ in rule R2 of Table 2 and by adding rule R6: $\langle \sum_{i=1}^{n} A_i \, , \, \sigma \rangle \longrightarrow_{T'} \langle A_i \, , \, \sigma \rangle$. This correctly describes the operational semantics of local choice in *ccp*. This is slightly different with respect to what is done in [JSS91], where the authors consider a rule for *global choice*. By defining the operational semantics $\mathcal{O}'$ according to the new transition system, we obtain the following result.

Proposition 1
$\mathcal{O}_D[\![A]\!](c) \subseteq \mathcal{O}'[\![A]\!](c) \subseteq Down(\mathcal{O}'_D[\![A]\!])(c) = \mathcal{N}[\![D.A]\!](\downarrow c)$

Proof of Proposition 1 First inclusion is easily obtained by examining transition systems T and T'. All the terminating configurations of T are terminating configurations for T' also, but due to the local-choice rule R6, there can be suspended configurations for T' not occurring in T. The second inclusion follows from the downward-closure definition.

The equivalence with the denotational semantics is obtained by induction on the number of procedure-call reductions in a computation, and on the form of the agent. In the parallel composition operator, downward closure allows us to assume the *restartability* of the processes.

Proof of Proposition 1 □

5 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 if and only if (for each initial store) the observables of the program are a subset of the property, i.e., for

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

Table 3: The finite angelic semantic operators

all $c \in C \,.\, \mathcal{O}_D[\![\, A \,]\!](c) \subseteq \phi$. Following this general view, we can formalize the static analysis of *cc* programs as a finite construction of an approximation (a superset) of 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* if and only if it is downward closed or upward closed. Ordering-closed properties are easier to verify, as shown by the following straightforward proposition.

Proposition 2 *A program $P = D.A$ satisfies a downward-closed (upward-closed) property $\phi \subseteq C$ if and only if 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 a 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 $d \vdash c$. Similarly, *freeness*[4] is an example of an upward-closed property.

[4]A variable x is free in $c \neq false$ if and only if $\exists_x c \neq c$. We assume no variable is free in $false$.

If x is free in c, then it is free in all the constraints d such that $c \vdash d$.

The framework of abstract interpretation, introduced by Cousot and Cousot [CC77, CC79b], is a powerful tool for the analysis of ordering preserving properties. Abstract interpretation is traditionally defined in terms of a pair of adjoint functions, called the *Galois connection*, which relates the concrete and abstract semantic domains (see [CC79b]). Galois connections here ensure the existence of the *best* approximations for both concrete objects and semantic functions, and provide a powerful tool for comparing the accuracy of different abstract semantics. In the following we consider downward-closed properties.

Definition 5 (Upper Galois Insertion) *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*

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

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).

This definition implies that both α (the abstraction function) and γ (the concretization function) are monotonic. As a matter of fact, α is a *complete join-morphism* and γ is a *complete meet-morphism*, and each one determines the other; i.e., $\alpha(x) = \sqcap' \{ y \in M' \mid x \leq \gamma(y) \}$ and $\gamma(y) = \bigsqcup \{ x \in M \mid \alpha(x) \leq' y \}$. Moreover, $\rho = (\gamma \circ \alpha)$ is an upper-closure operator on M, mapping each concrete object to its upper approximation [CC79b].

Upper Galois insertions are commonly used in abstract interpretation of (constraint) logic languages. Here the approximation process returns weaker (with regard to $\vdash$) semantic objects (an example for the *clp* case is in [GDL95]). Lower Galois insertions are defined dually. They induce an approximation process returning stronger semantic objects which can be used to approximate the "maximal" guarantee of a program. An example of lower Galois insertions for polymorphic typing is in [Mon92]. In the following, we will consider the more "standard" upper insertions only.

6 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'.

6.1 Relating Constraint Systems

In this section we formalize the notion of *correct upper approximation* between constraint systems.

Definition 6 (Correctness) *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 $\alpha : C \to \mathcal{A}$, if and only if (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 3 *If $\mathcal{A}$ is upper correct with regard to constraint system C using α, then there exists an upper Galois insertion relating $\mathcal{P}{\downarrow}(C)$ and $\mathcal{P}{\downarrow}(\mathcal{A})$.*

Proof of Proposition 3 Consider $\langle \mathcal{P}{\downarrow}(C), \tilde{\alpha}, \gamma, \mathcal{P}{\downarrow}(\mathcal{A}) \rangle$, where

$$\tilde{\alpha}(S) = \{\alpha(c) \in \mathcal{A} \mid c \in S\} \quad \text{and} \quad \gamma(S') = \cup\{T \in \mathcal{P}{\downarrow}(C) \mid \tilde{\alpha}(T) \subseteq S'\}$$

Linearity of $\tilde{\alpha}$ implies additivity, because in this case set union is also the *lub* of the lattices. Moreover α-surjectivity on $\mathcal{A}$ implies $\tilde{\alpha}$-surjectivity on $\mathcal{P}{\downarrow}(\mathcal{A})$. The proof is complete, since any additive and surjective function between complete lattices defines a Galois insertion [CC79b].

Proof of Proposition 3 $\square$

The following corollary is a consequence of α monotonicity.

Corollary 1 *If $\mathcal{A}$ is upper correct with respect to the constraint system C using α, then for all $c, d \in C$ we have $\alpha(c \otimes d) \vdash' \alpha(c) \otimes' \alpha(d)$.*

The following definition states a property that intuitively holds for all meaningful upper-correct constraint systems (the same property was considered in [GDL95] for *uco* on constraint systems).

Definition 7 ($\exists$-α Confluence) *A constraint system $\mathcal{A}$ upper correct with regard to C using α satisfies $\exists$-α confluence if and only if, for all $x \in V$, $c \in C$, $\exists'_x \alpha(\exists_x c) = \alpha(\exists_x c)$.*

This simply means that, given a constraint having no information on the variable x (i.e., $\exists_x c$), the abstraction process cannot produce information on x. As mentioned before, this is intuitively true, because abstraction corresponds to weakening of constraints.

Proposition 4 *If $\mathcal{A}$ is upper correct with regard to the constraint system C using α and satisfying $\exists$-α confluence, then for all $x \in V$, $c \in C$, $\exists'_x \alpha(c) = \alpha(\exists_x c)$.*

Proof of Proposition 4 By $\exists$ extensivity, α and $\exists'$ monotonicity, and confluence, we have

$$c \vdash \exists_x c \Rightarrow \alpha(c) \vdash' \alpha(\exists_x c) \Rightarrow \exists'_x \alpha(c) \vdash' \exists'_x \alpha(\exists_x c) = \alpha(\exists_x c)$$

Correctness completes the proof.

Proof of Proposition 4 $\square$

Example 3 (Relating Herbrand and $\boldsymbol{DEP_g}$) *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.*

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

Proposition 5 *The constraint system DEP_g is upper correct with regard to C_H using α_g. Moreover, DEP_g satisfies the $\exists$-α_g confluence.*

To guarantee the sure termination of the analysis, we consider finite abstract constraint systems only.

6.2 The Abstract Synchronization Problem

Let us consider the angelic concurrent language, and let f be a linear lower-closure operator on $\mathcal{P}{\downarrow}(C)$ (i.e., the concrete semantics of an agent). Let $\mathcal{A}$ be an abstract constraint system upper correct with regard to 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. Then f' is a *correct* upper approximation of f on $\mathcal{P}{\downarrow}(\mathcal{A})$ if and only if $f^\sharp \vdash' f'$ [CC79b].

Proposition 6 *$f^\sharp = (\tilde{\alpha} \circ f \circ \gamma)$ is a linear lco on $\mathcal{P}{\downarrow}(\mathcal{A})$.*

Proof of Proposition 6 It is straightforward to see that $f^\sharp$ is a lower-closure operator on $\mathcal{P}{\downarrow}(\mathcal{A})$. It is also linear, since it is the composition of three linear functions.

Proof of Proposition 6 □

The abstract and concrete semantics of angelic processes can be modeled in the same way. However, the simple transformation considered in [GDL95] is no longer admissible for *cc* programs, because the abstract synchronization operator is not correct. The following theorem justifies this observation.

Theorem 1

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

Proof of Theorem 1 The left arrow is straightforward. Let $\rho = (\tilde{\alpha} \circ \gamma)$. α is an isomorphism if and only if ρ is the identity function for $\mathcal{P}{\downarrow}(C)$. Suppose α is not an isomorphism. Thus there exists $c \in C$ such that $({\downarrow}c) \neq \rho({\downarrow}c)$. Since ρ is a *uco* on $\mathcal{P}{\downarrow}(C)$, this means $\rho({\downarrow}c) \not\subseteq ({\downarrow}c)$, i.e., there exists a $\tilde{c} \in \rho({\downarrow}c)$ such that $\tilde{c} \notin ({\downarrow}c)$. Consider the synchronization operator $\mathbf{ask}(c) \to f$. We have $[\mathbf{ask}(c) \to f]^\sharp({\downarrow}\alpha(c)) = {\downarrow}\alpha(c)$, because (by linearity) the best correct synchronization test for $\tilde{c}$ (i.e., ${\downarrow}\tilde{c} \subseteq {\downarrow}c$) is not satisfied.

On the other hand, $(\mathbf{ask}(\alpha(c)) \to f')({\downarrow}\alpha(c)) = f'({\downarrow}\alpha(c))$, because the abstract synchronization test (i.e., ${\downarrow}\alpha(c) \subseteq {\downarrow}\alpha(c)$) is always satisfied.

Since f' is reductive, in the general case we have $\downarrow\alpha(c) \not\subseteq f'(\downarrow\alpha(c))$. Thus we have lost correctness.

Proof of Theorem 1 □

The above result specifies that the traditional form of abstraction of constraint logic languages implemented in [CF92, GDL95] is no longer applicable to *ccp* programs.

6.3 An Easy Solution: Removing Synchronizations

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 behavior.

Definition 8 *The success semantics for program $P = D.A$ is given by the function:*

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

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

The next (straightforward) proposition justifies our interest in such a semantic definition, and motivates further research in designing accurate suspension-freeness analyses.

Proposition 7 *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 to have a correct abstract synchronization operator, we must grant the following condition:

$$\text{concrete computation proceeds} \;\Rightarrow\; \text{abstract computation proceeds.}$$

Thus, whenever we cannot prove the contrary, we assume that the concrete computation proceeds.

The simplest way to satisfy the previous correctness condition consists in removing all synchronizations from the program. Consider the transformation $NoSynch$: Program $\to$ Program defined in Table 4. Let $\tilde{P} = \tilde{D}.\tilde{A} = NoSynch[P]$. Since we have discarded every meaningful synchronization test, processes in the transformed program $\tilde{P}$ always proceed, providing a correct approximation of the success semantics of the original program $P = D.A$.

$NoSynch[Dec.A] = NoSynch[Dec].NoSynch[A]$
$NoSynch[\epsilon] = \epsilon$
$NoSynch[p(\vec{x}) :- A.Dec] = p(\vec{x}) :- NoSynch[A].NoSynch[Dec]$
$NoSynch[\mathbf{tell}(c)] = \mathbf{tell}(c)$
$NoSynch[\exists\vec{x}.A] = \exists\vec{x}.NoSynch[A]$
$NoSynch[A \parallel B] = NoSynch[A] \parallel NoSynch[B]$
$NoSynch[\sum_{i=1}^{n} (\mathbf{ask}(c_i) \rightarrow A_i)] = \bigoplus_{i=1}^{n} (\mathbf{tell}(c_i) \parallel NoSynch[A_i])$
$NoSynch[p(\vec{y})] = p(\vec{y})$

Table 4: The transformation *NoSynch*

Proposition 8 *For all $c \in C$, we have $\mathcal{SS}_D[\![A]\!](c) \subseteq \mathcal{SS}_{\tilde{D}}[\![\tilde{A}]\!](c)$.*

Transformed programs are very similar to sequential constraint logic programs. Since processes do not synchronize anymore, their semantics can easily be modeled by a single (possibly disjunctive) constraint. Following [FLMP89, GDL95], we define a fixpoint semantics for the transformed programs that is proved equivalent to the *downward closure* of the success semantics.[5] Diagonal elements, cylindrification operators, and special variables ψ_i provide the independence from variable names.

In the following, we (re-)define the semantics of *NoSynch*-transformed programs in terms of a single predicate transformer, in the style of standard *clp* semantics. Clearly, closure-operator-based semantics still work for these kinds of programs.

Definition 9 (C-Interpretation) *Let C be a constraint system. A constrained atom has the form $p(\vec{\psi}) :- S$, where $S \in \mathcal{P}{\downarrow}(C)$ and $FV(S) \subseteq \vec{\psi}$. Let $\mathcal{B}$ be the set of constrained atoms defined over an alphabet of process identifiers Π_D. We define the partial order $\preceq$ on $\mathcal{B}$ such that $p(\vec{\psi}) :- S_1 \preceq p(\vec{\psi}) :- S_2$ if and only if $S_1 \subseteq S_2$. The set $\mathcal{B}$ is the base of interpretations. An interpretation is any subset of $\mathcal{B}$. We denote by $\Im \subseteq \wp_f(\mathcal{B})$ the family of C-interpretations, i.e., the interpretations containing at most one constrained atom for each process identifier. The partial order defined on $\mathcal{B}$ is naturally extended to $\Im$.*

[5] Using the S-semantics approach [FLMP89], it is also possible to give a fixpoint semantics equivalent to the success semantics [GDL95]. Note, however, that for downward-closed program properties this difference is not meaningful.

Proposition 9 $(\Im, \preceq)$ *is a complete lattice.*

In the following, P is a *NoSynch*-closed C-program (i.e., $P = NoSynch[P]$). Moreover, we assume that a program contains a single clause for each predicate. This can be easily obtained by joining all definitions for a predicate in the body of a single clause. The fixpoint semantics is defined in terms of an immediate-consequences operator, or predicate transformer, T_P^C.

Definition 10 *The mapping* $T_P^C : \Im \rightarrow \Im$ *is defined as follows:*

$$T_P^C(I) = \left\{ p(\vec{\psi}) :- S \;\middle|\; p(\vec{x}) :- A \in P \;,\; S = \exists_{\vec{x}}((\downarrow d_{\vec{x}\vec{\psi}}) \cap \mathcal{E}[\![A]\!] I) \right\}$$

where

- $\mathcal{E}[\![\mathbf{tell}(c)]\!] I = \downarrow c$,
- $\mathcal{E}[\![\exists \vec{x}.A]\!] I = \exists_{\vec{x}} \mathcal{E}[\![A]\!] I$,
- $\mathcal{E}[\![A \parallel B]\!] I = \mathcal{E}[\![A]\!] I \cap \mathcal{E}[\![B]\!] I$,
- $\mathcal{E}[\![\bigoplus_{i=1}^{n} A_i]\!] I = \bigcup_{i=1}^{n} \mathcal{E}[\![A_i]\!] I$, *and*
- $\mathcal{E}[\![p(\vec{y})]\!] I = \begin{cases} \exists_{\vec{\psi}}(\downarrow d_{\vec{\psi}\vec{y}} \cap S) & \text{if } p(\vec{\psi}) :- S \in I \\ \{false\} & \text{otherwise.} \end{cases}$

Clearly, T_P^C is a continuous function on the complete lattice $(\Im, \preceq)$. Hence we can define a fixpoint semantics $\mathcal{F}^C(P) = lfp(T_P^C) = T_P^C \uparrow \omega(\emptyset)$.

Theorem 2 *Let* $P = D.A$ *be a NoSynch-closed program. If* $\mathcal{SS}_D[\![A]\!] \neq \emptyset$ *then* $Down(\mathcal{SS}_D[\![A]\!]) = \lambda c \in C.(\downarrow c) \cap \mathcal{E}[\![A]\!](\mathcal{F}^C(P))$.

Proof of Theorem 2 Since in program P there are no meaningful synchronizations, the behavior of processes cannot be influenced by the external environment. As a consequence, we only have to prove that $\downarrow(\mathcal{SS}_D[\![A]\!](true)) = \mathcal{E}[\![A]\!](\mathcal{F}^C(P))$. This is done by induction on the number of procedure-call reductions in a computation, and on the agent form.

Proof of Theorem 2 $\square$

The abstract semantics of the transformed program (obtained by replacing the concrete constraints by the corresponding abstract constraints) is a correct approximation of its concrete semantics. For $I \in \Im$, let us define $\alpha(I) = \left\{ p(\vec{\psi}) :- \tilde{\alpha}(S) \,\middle|\, p(\vec{\psi}) :- S \in I \right\}$.

Theorem 3 *Let P be a NoSynch-closed C-program, and let P' be the corresponding abstract program on $\mathcal{A} = \alpha(C)$. Then $\alpha(\mathcal{F}^C(P)) \preceq' \mathcal{F}^{\mathcal{A}}(P')$.*

Example 4 *Consider the program D 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)
```

The transformed program $\tilde{D} = NoSynch[D]$ (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 $\mathcal{A} = Prop$. The abstract program P' on Prop corresponding to $\tilde{P}$ is:

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

By computing the semantics of P', we obtain:

$$\mathcal{F}^{\mathcal{A}}(P') = \left\{ \ app(\psi_1, \psi_2, \psi_3) :- (\psi_1 \wedge \psi_2) \leftrightarrow \psi_3 \ \right\}$$

Thus, in all the answer constraints associated to successful computations of the original program, the third argument of app is bound to a ground term if and only if both the first and the second argument are bound to ground terms.

7 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:

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

Clearly, the test is here based on the concretization function γ. It is easy to see that, by the standard properties of Galois insertions, this test cannot be verified by looking at the abstract values only, i.e., for any abstract and concrete constraints, respectively a and c, $\gamma({\downarrow}a) \subseteq \gamma(\alpha({\downarrow}c)) \not\Rightarrow \gamma({\downarrow}a) \subseteq ({\downarrow}c)$. Therefore, the above test may involve a computation over a possibly infinite set: the concrete domain.

In practice, we have to implement a "hybrid" synchronization test that verifies whether an abstract constraint *definitely entails* a concrete one:

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

Note that a similar hybrid test has been introduced in [FGMP93]. Informally, this condition means "if the abstract computation proceeds, then every concrete computation it approximates proceeds too."

Remark 1
Static analysis by Angel program transformation cannot be considered as being based on generalized semantics. This is because the abstract program does not perform all computations on the abstract constraint system.

Now suppose we have found a meaningful (i.e., useful in practice) synchronization primitive. Next, we have to choose a suitable approximation of the nondeterministic operator. To get an efficient abstract interpretation framework, we cannot directly abstract global choice, since the associated denotational models are too complex [SRP91]. Local choice (i.e., angelic languages) seems to be a good cost/precision trade-off. We call this form of synchronization *condensed*.

Consider the transformation *Angel*, mapping arbitrary *cc* programs into angelic *cc* programs with condensed synchronization. The denotational semantics of this kind of program can be obtained by using Table 3 and by replacing the equation for the (simple) synchronization operator with the following equation:

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

$$
\begin{array}{|l|}
\hline
Angel[Def.A] = Angel[Def].Angel[A] \\
Angel[\epsilon] = \epsilon \\
Angel[p(\vec{x}) :- A.Def] = p(\vec{x}) :- Angel[A].Angel[Def] \\
Angel[\mathbf{tell}(c)] = \mathbf{tell}(c) \\
Angel[\exists\vec{x}.A] = \exists\vec{x}.Angel[A] \\
Angel[A \parallel B] = Angel[A] \parallel Angel[B] \\
Angel[\sum_{i=1}^{n}(\mathbf{ask}(c_i) \rightarrow A_i)] = \mathbf{ask}(c_1; \ldots; c_n) \rightarrow \bigoplus_{i=1}^{n} (\mathbf{tell}(c_i) \parallel Angel[A_i]) \\
Angel[p(\vec{x})] = p(\vec{x}) \\
\hline
\end{array}
$$

Table 5: The transformation *Angel*

The meaning of the condensed synchronization test is to **ask** the *disjunction* (on $\mathcal{P}{\downarrow}(C)$) of all the guard constraints

$$\forall \sigma \in S \ . \ \exists j \in \{1, \ldots, n\} \ . \ ({\downarrow}\sigma) \subseteq ({\downarrow}c_j) \quad \Leftrightarrow \quad S \subseteq \bigcup_{i=1}^{n} ({\downarrow}c_i)$$

Remark 2 *This is not true when we consider a widening as a disjunction operator. As an example, consider a constraint system dealing with rational intervals with entailment given by inclusion. Consider the following multiple* ***ask*** *and its widened version:*

$$\mathbf{ask}(x \in [0,1]; x \in [1,2]) \rightarrow A$$
$$\mathbf{ask}(x \in [0,2]) \rightarrow A$$

Given the initial store $x \in [0,2]$, the first computation (correctly) suspends, while the latter proceeds, possibly providing incorrect results.

Therefore, given $S^\sharp \in \mathcal{P}{\downarrow}(\mathcal{A})$ and $c_1, \ldots, c_n \in C$, the *condensed abstract synchronization test* is defined as:

$$mtest(S^\sharp, c_1; \ldots; c_n) = true \quad \Rightarrow \quad \forall \sigma \in \gamma(S^\sharp) \ . \ \exists i_\sigma \in \{1, \ldots, n\} \ . \ \sigma \vdash c_{i_\sigma}$$

Note that the index i_σ depends on σ. This simply means that different stores possibly satisfy different guard constraints. Indeed, there can be suspension-free choice operators having no definitely satisfied guards (e.g., deterministic choice operators).

The following example illustrates a suspension-freeness analysis for a common communication scheme.

Example 5 *In the following simple program [Sha89], a producer* pzaff *sends messages to different 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)
```

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***:*

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

For this purpose, the rigid types abstraction, discussed in [JB92] and further used for the systematic derivation of norms for termination analysis of logic programs in [DSF93], 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, this 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).*

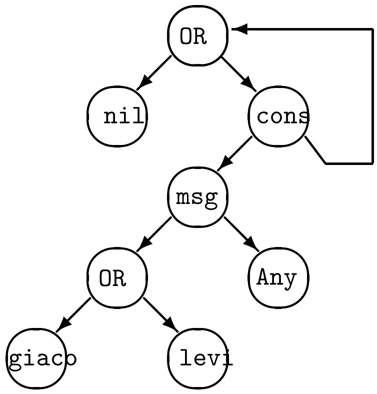

Figure 1: The rigid typegraph for X

The following theorem relates the standard, angelic, and success semantics of a *cc* program (on the concrete constraint system). In particular, for suspension-free programs *NoSynch* is always better than *Angel*, since the latter can suspend when approximating synchronization.

Theorem 4 *Given* $P = D.A$*, let* $P_1 = D_1.A_1 = NoSynch[P]$ *and* $P_2 = D_2.A_2 = Angel[P]$*:*

- $\mathcal{O}_D[\![A]\!](c) \subseteq \mathcal{N}[\![D_2.A_2]\!](\downarrow c)$*, and*
- *if* P *is suspension-free, then*

$$\mathcal{O}_D[\![A]\!](c) \subseteq \mathcal{SS}_{D_1}[\![A_1]\!](c) \subseteq (\downarrow c) \cap \mathcal{E}[\![A_1]\!](\mathcal{F}^C(D_1)) \subseteq \mathcal{N}[\![D_2.A_2]\!](\downarrow c)$$

Proof of Theorem 4 To prove the first statement, we observe that every terminating computation in the transition system of P has a corresponding terminating computation in the transition system of P_2, producing the same answer constraint, i.e., $\mathcal{O}_D[\![A]\!](c) \subseteq \mathcal{O}_{D_2}[\![A_2]\!](c)$. Then, we apply Proposition 1.

If P is suspension-free, by Propositions 7 and 8 we obtain first inclusion. Second inclusion follows from Theorem 2, and the third one is obtained by observing that P_2 has the same successful computations of P_1, but it can still suspend.

Proof of Theorem 4 □

$$\begin{array}{l}
\mathcal{E}' : Agent \times Env \to llco(\mathcal{P}{\downarrow}(\mathcal{A})) \\
\mathcal{E}'[\![\,\mathbf{tell}(\alpha(c))\,]\!]e = {\downarrow}\alpha(c) \\
\mathcal{E}'[\![\,\mathbf{ask}(c_1; \ldots; c_n) \to A\,]\!]e = \\
\qquad\qquad \{a \in A \mid mtest({\downarrow}a, c_1; \ldots; c_n) = true \;\Rightarrow\; a \in \mathcal{E}'[\![\,A\,]\!]e\} \\
\mathcal{E}[\![\,\exists\vec{x}.A\,]\!]e = \{a' \in \mathcal{A} \mid \text{there exists } a \in \mathcal{E}'[\![\,A\,]\!]e \text{ s.t. } \exists_{\vec{x}}a = \exists_{\vec{x}}a'\} \\
\mathcal{E}'[\![\,A \parallel B\,]\!]e = \mathcal{E}'[\![\,A\,]\!]e \;\cap\; \mathcal{E}'[\![\,B\,]\!]e \\
\mathcal{E}'[\![\,\bigoplus_{i=1}^{n} A_i\,]\!]e = \bigcup_{i=1}^{n} \mathcal{E}'[\![\,A_i\,]\!]e \\
\mathcal{E}'[\![\,p(\vec{y})\,]\!]e = \left\{a' \in \mathcal{A} \;\middle|\; a' = \exists'_{\vec{\psi}}(d'_{\vec{y}\vec{\psi}} \otimes' a),\, a \in (e\ p)\right\} \\
\mathcal{D}' : Dec \times Env \to Env \\
\mathcal{D}'[\![\,\epsilon\,]\!]e = e \\
\mathcal{D}'[\![\,p(\vec{x}) :-\ A\,.\,D\,]\!]e = \mathcal{D}'[\![\,D\,]\!]\left(e\left[p \mapsto \mathcal{E}'[\![\,\exists\vec{x}.(\mathbf{tell}(d'_{\vec{\psi}\vec{x}}) \parallel A)\,]\!]e\right]\right) \\
\mathcal{N}' : Progr \to llco(\mathcal{P}{\downarrow}(\mathcal{A})) \\
\mathcal{N}'[\![\,D.A\,]\!] = \mathcal{E}'[\![\,A\,]\!](lfp\,\mathcal{D}'[\![\,D\,]\!])
\end{array}$$

Table 6: The abstract angelic semantic operators

The same situation occurs when considering the abstract semantics construction, provided that we have defined a specific abstract synchronization test and proved it correct (see Table 6).

Theorem 5 *Given P, let $P'_1 = \alpha(NoSynch[P])$, and let P'_2 be the program obtained by replacing all the* **tell** *constraints in $Angel[P]$ by the corresponding abstractions:*

- *$\mathcal{N}'[\![\,P'_2\,]\!]$ is a correct abstraction of $\mathcal{O}[\![\,P\,]\!]$, and*
- *P is suspension-free $\Rightarrow \mathcal{F}^{\mathcal{A}}(D'_1)$ is correct with regard to $\mathcal{O}[\![\,P\,]\!]$, and gives better results than $\mathcal{N}'[\![\,P'_2\,]\!]$.*

Proof of Theorem 5 Just modify the proof of Theorem 4 by taking into account correctness of the abstract constraint system and the condensed abstract synchronization test.

Proof of Theorem 5 □

For any suspension-free program P, the abstract interpretation based on the transformation *Angel*returns the same result of the abstract interpretation based on the transformation *NoSynch* only when the former can "prove" suspension freeness.

8 Related Works

In earlier concurrent logic languages, the semantics was given in operational style, since no clear declarative reading of the synchronization mechanism was available. Therefore the initial approaches to static analysis were based on the operational semantics. In particular, [CCC90] defines a scheme for the detection of suspension-free FCP(:) programs. The analysis is an abstraction of the AND-OR tree operational model defined in [CF89]. The same problem is addressed in [CFM94], where the analysis of FCP(:) programs is achieved by abstracting the transition-system operational semantics. It is also shown how to obtain analyses for local suspension, deadlock, and local deadlock. Later, this approach was extended to *cc* languages with consistency check [CFMW93]. By using the abstract domain DEP_g together with suitable semilinear norms, it is possible to infer suspension freeness of some producer-consumer programming scheme. However, in [CFMW93] the correctness of the abstract synchronization test lies in the consistency check. When dealing with the language defined in [SRP91] (i.e., without the consistency check), this abstract test is no longer correct. To get independence from the scheduling policy, [CFM94] and [CFMW93] use a nonstandard (operational) semantics that makes the computation confluent. This approach has inspired our program transformation *Angel*, which can be seen as the denotational translation of the confluent transition system.

To our knowledge, [FGMP93] defines the first abstract interpretation framework for *cc* programs based on a denotational (and compositional) semantics. Also, 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. As the authors of [FGMP93] admit, 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). Clearly, in this approach, one might get additional suspensions, but still all successful computations are preserved.

The problem of giving a generalized abstract interpretation framework for *cc* languages, where only local choice is allowed, is considered in [CC93]. However, in contrast with Theorem 1, they claim that it is *correct* to directly abstract the program, in the style of [CF92, GDL95], and evaluate it on the abstract constraint system. This is clearly in contrast to our result, where we proved that it is *not* sound to "translate" the approach in [CF92, GDL95] to the analysis of *ccp*. In general, this approach to static analysis returns incorrect results because of the abstract synchronization problem.

A more recent paper, [FGMP95], considers the analysis of compositionally confluent *cc* programs and defines an abstract interpretation framework that is very similar to that obtained by our transformation *Angel*. This approach, based on the denotational semantics of angelic *cc*, maps each (nonconfluent) 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 $\mathcal{P}{\downarrow}(C)$ (see Remark 2). The only difference is that, once the synchronization test is passed, this transformation does not use the guard constraints to strengthen each branch of the computation. On the contrary, *Angel* tells each branch's guard before proceeding in the abstract computation, possibly obtaining stronger (better) results.

9 Conclusions

We have shown that the **ask** operators cannot be safely *upper* approximated employing the traditional methods for semantics approximation used in sequential constraint logic programs. 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 [BCGL92], the semantics of (pure) Prolog programs (logic programs with depth-first search) can be specified in terms 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 behavior is also shared by semantic models for built-ins in Prolog [AMP92]. While the *implicit ask-mapping-based* semantic definitions for Prolog's search or built-ins 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.

We are currently investigating other kinds of approximations. In particular, we believe that the **ask** operators allow the use of a generalized semantics approach when we deal with *lower* approximations. In this case, we obtain information about the *definite nonentailment* of guard constraints, allowing the pruning of useless branches of the computation.

Acknowledgement of support: The work of E. Zaffanella and G. Levi has been supported by the PARFORCE (Parallel Formal Computing Environment) BRA-Esprit II Project 6707. The work of R. Giacobazzi has been partially supported by the Project inter-PRC: Langages Logiques Concurrents avec Contraintes and by the EEC–HCM individual grant: Semantic Definitions, Abstract Interpretation and Constraint Reasoning, ERBCHBICT930822. This work was performed while R. Giacobazzi was visiting LIX, Laboratoire d'Informatique, École Polytechnique, Paris. He thanks LIX, and in particular Radhia Cousot, for the kind hospitality.

References

[AMP92] K. R. Apt, E. Marchiori, and C. Palamidessi. A theory of first-order built-ins of PROLOG. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 69–83, Berlin, 1992. Springer-Verlag.

[BCGL92] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog control. In *Proceedings of the Nineteenth Annual ACM*

Symposium on Principles of Programming Languages, pages 95–104, New York, 1992. ACM Press.

[Bir67] G. Birkhoff. *Lattice Theory.* AMS Colloquium Publications, 3rd edition, 1967.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.

[CC79a] 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.

[CC79b] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, 1979. ACM Press.

[CC93] 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.

[CCC90] C. Codognet, P. Codognet, and M. Corsini. Abstract interpretation for concurrent logic languages. In S. K. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming '90*, pages 215–232, Cambridge, MA, 1990. The MIT Press.

[CF89] M. Corsini and G. Filè. A complete framework for the abstract interpretation of logic programs: theory and application. Technical report, Università di Padova, Italy, 1989.

[CF92] P. Codognet and G. Filè. Computations, abstractions and constraints. In *Proceedings of the Fourth IEEE International Con-*

ference on Computer Languages, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[CFM94] M. Codish, M. Falaschi, and K. Marriott. Suspension analyses for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.

[CFMW93] 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, *Proceedings of the 20th International Colloquium on Automata, Languages, and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 633–644, Berlin, 1993. Springer-Verlag.

[dBP91] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T. Maibaum, editors, *Proceedings of TAPSOFT '91*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319, Berlin, 1991. Springer-Verlag.

[DSF93] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, Cambridge, MA, 1993. The MIT Press.

[FGMP93] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[FGMP95] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and concurrent constraint programming. In *Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, volume 936 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer-Verlag.

[FLMP89] 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.

[GDL95] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–248, 1995.

[HMT71] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras. Part I.* North-Holland, Amsterdam, 1971.

[JB92] 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.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, New York, 1987. ACM Press.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic nondeterminism in concurrent constraint programming. Technical report, System Science Lab., Xerox PARC, 1991.

[Mah87] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[Mon92] B. Monsuez. Polymorphic typing by abstract interpretation. In R. Shyamasundar, editor, *Proceedings of the 12th International Conference on FST&TCS*, volume 652 of *Lecture Notes in Computer Science*, pages 217–228, Berlin, 1992. Springer-Verlag.

[Mor60] J. Morgado. Some results on the closure operators of partially ordered sets. *Portugaliæ Mathematica*, 19(2):101–139, 1960.

[Sco82] D. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Proceedings of the Ninth International Colloquium on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613, Berlin, 1982. Springer-Verlag.

[Sha89] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[SR90] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245, New York, 1990. ACM Press.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of concurrent constraint programming. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–353, New York, 1991. ACM Press.

[War42] M. Ward. The closure operators of a lattice. *Annals of Mathematics*, 43(2):191–196, 1942.