Angelic non-determinism in concurrent constraint programming

Radha Jagadeesan
Imperial College,

Vijay A. Saraswat
Xerox PARC

Vasant Shanbhogue
Wichita State University

Abstract

Concurrent constraint programming [Sar89,SR90,SRP91] is a simple and powerful model of computation based on the notions of *store-as-constraint* and *process as information transducer*. In this paper we describe a (domain-theoretic) semantic foundation for cc languages with ask, tell, parallel composition, hiding, recursion and angelic non-determinism ("parallel backtracking"). This class of languages includes the cc/Herbrand language [Sar89] and a simpler reworking [HS91] of CHIP.

Generalizing previous work on determinate constraint programming [JPP89,SRP91], we describe the semantics for such a language based on modelling a process as a set of constraints, with parallel composition (conjunction) given by set intersection and or-parallel search (disjunction) given by set union. This is achieved by viewing processes as continuous linear closure operators on the Smyth power-domain of the underlying constraint system. The model is shown to be fully abstract for the observation of finite approximants to the limits of fair execution sequences. We also give a model for the "non-ground success set" semantics—a model fully abstract for the observation of the minimal final stores in terminated execution sequences.

1 Introduction

Concurrent constraint programming [Sar89,SR90,SRP91] arises from earlier work in constraint logic programming [JL87] and concurrent logic programming [Mah87,Sar87a]. The basic move in this paradigm is to replace the notion of store-as-valuation central to von Neumann computing with the notion that the store is a constraint, that is, pieces of partial information about the values that variables can take. The usual notions of "read" and "write" no longer make sense; instead computation progresses via the execution of *ask* and *tell* actions. An ask operation takes a constraint (say, c) and uses it to probe the structure of the store: it succeeds if the store contains enough information to entail c. A tell operation takes a constraint and conjoins it with the constraints already in the store. Thus computation progresses via the monotonic accumulation of information. Concurrency arises because any number of agents may simultaneously interact with the store. Synchronization is achieved via a blocking ask—an agent blocks if the store is not strong enough to entail the constraint it wishes to check; it remains blocked until such time as (if ever) some other concurrently executing agents add enough information to the store for it to be strong enough to entail the query. Finally, to asbtract away from timing considerations, tells are executed asynchronously: thus, it is guaranteed that the store eventually satisfies the constraint, but no assumptions are made on the amount of time taken to achieve this result.

This paper is concerned with laying the semantic foundations for what we shall call the *Angelic* cc languages—the basic primitives provided are ask, tell, parallel composition, hiding, recursion and non-deterministic disjunction. Non-determinism arises, as in logic programming, because an agent A is allowed to decompose *disjunctively* into one or more agents, say $A_1, \ldots, A_n$. In such a case, n copies of the current store are made, and the ith copy is associated with an agent A_i and a copy of all the other agents interacting with the original store (which are thus also replicated n times). The task of the computation is now to find (one or all) the resultant *consistent* stores reachable from the initial store. The non-determinism is termed angelic, because it is *error*-avoiding, in a sense similar to (constraint) logic programming: if an execution branch leads to (logical) inconsistency, then this branch is omitted (i.e., "backtracked on", though it must be kept in mind that all or-parallel branches are conceptually being explored simultaneously, so "backtracking" is a misnomer). The non-determinism is *not* deadlock-avoiding: if one branch of the computation is unable to progress because all the agents on the branch are ask-agents, then the store associated with this branch is thought of as one possible result of the computation. Note that this is in step with the spirit of asynchronous computation; the idea is to insulate the programmer from timing considerations. [1] Thus, deadlock is identified with *non-failing termination*, not with inconsistency.

While such non-deterministic splitting can be quite expensive to implement in a distributed context, it is invaluable in various applications primarily concerned with combinatorial search. Combined with the ask-and-tell synchronization mechanism discussed above, this kind of non-determinism provides a formulation of constraint programming which is very useful in writing sophisticated search programs in very simple ways. The basic point is that explicit synchronization allows the programmer to exercise a much finer control over the execution of a non-deterministic program than is possible in, say logic programming languages (such as pure Prolog). From a programming point of view, this form of control has been explored in detail in earlier work (e.g., [Sar87b,Sar89,SRH91]). It has also been embedded within other non-deterministic concurrent programming languages such as cc/Herbrand [Sar89], and [HS91], which provides a "rational reconstruction" (and extension) of CHIP [DvH$^+$88]. The semantic treatment presented in this paper serves to provide a simple mathematical foundation for these languages. Note that determinate constraint programming [SRP91] and (constraint) logic programming can be viewed as specializations of the Angelic cc framework. To obtain determinate cc languages, disallow disjunctions; to obtain constraint logic progamming, disallow ask operations; to obtain (plain) logic programming take the special case in which the constraint system is Herbrand [Sar89]. search process.

The main contribution of this paper isa domain theoretic account of the process of search. Previous work [JPP89,SRP91] showed that determinate processes can be thought of as denoting *closure operators* on the underlying constraint system. Thus, a process was characterized by the set of its *resting-points*—the stores on which the process quiesces without producing more information. In this paper, we show that *exactly the same information* is needed for characterizing Angelic cc processes. In particular, no other information (e.g. about execution paths) needs to be recorded to capture the effects of non-determinism. The underlying reason for this very simple model, henceforth called the $\mathcal{N}$-model, is that a program can be taken to denote a *continuous linear closure operator over the Smyth powerdomain* $\mathcal{P}_I(E)$ of the underlying constraint system E. We prove that such operators can be characterized by the set of their singleton fixed-points. This leads to a very simple semantics for and-parallel composition as intersection and or-parallel composition as union of these sets of constraints. We also give an operational semantics which observes finite approximants to limits of fair execution sequences. The model is shown to be fully-abstract for this semantics.

We show that minor variations of this basic idea describe a semantics—called the $\mathcal{S}$-model in what follows—which can be used to characterize the finite-success set of the program. This model captures the results produced by terminating execution sequences. The $\mathcal{S}$-model generalises existing models of determinate constraint programming and (constraint) logic programming. In particular, for (constraint) logic programs, the model yields the set of minimal answers [FLMP89,GS89]. This paper shows that the simple (constraint) logic programming view of programs discussed in these papers holds in the presence of synchronization. Furthermore, restricted to determinate programs, the semantics cuts down to the the semantics for determinate constraint programs, as discussed in earlier work [SRP91].

The rest of the paper is organised as follows. The next section uses a simple example to motivate the operational assumptions and the abstract view of programs. Next, we quickly sketch the operational semantics ($\mathcal{O}_\infty$) for the cc languages, following [SRP91]. In the next section, we present the $\mathcal{N}$-model for the language. We then establish the connection between $\mathcal{O}_\infty$ and $\mathcal{N}$. Finally, we describe the operational semantics corresponding to observing terminated execution sequences and also the denotational model $\mathcal{S}$, which corresponds to it.

2 Basic intuitions

Consider the process

$$((X = 1) \rightarrow (Y = 2)) \vee (Z = 3)$$

Operationally, this process can be read as: decompose into two branches. In the first, Ask if $X = 1$ (that is, suspend until the store in that branch has enough information to answer $X = 1$) and on success add $Y = 2$ to the store (in that branch). In the second, add the constraint $Z = 3$ to the store (in that branch). The set of resulting stores obtained by executing the process in the uninitialised store (that is, the store `true`)

[1] Note that if the store is inconsistent, then all ask-agents can reduce because the inconsistent constraint entails every other constraint. Hence a branch is not thought of as deadlocking if its associated store is inconsistent.

is `true` (given by the first disjunct) and the constraint $(Z = 3)$ (given by the second disjunct). In this case, the unique resulting minimal store is just `true`. Similarly there are two possible results from executing the process in a store with the unique constraint $(X = 1)$: namely, $(X = 1, Y = 2)$ and $(X = 1, Z = 3)$.

Angelic non-determinism and deadlock

Consider the above program run in parallel with $(X = 1)$: that is, the program

$$[((X = 1) \rightarrow (Y = 2)) \vee (Z = 3)] \wedge (X = 1)$$

started off in the initially empty store. If the search process were deadlock avoiding, there are two sets of possible stores depending on the relative speeds of the two conjuncts.

- If the right conjunct is executed faster, then the final results of the program are the following two stores: the store with the constraint $(X = 1, Y = 2)$ or the store with the constraint $(X = 1, Z = 3)$.
- If the left conjunct is executed faster, then the unique final result of the program is the store with the constraint $(X = 1, Z = 3)$.

If the non-determinism is not deadlock avoiding, the two possible resulting stores have the constraints $(X = 1, Z = 3)$ and $(X = 1, Y = 2)$. More importantly, these results are obtained without reasoning about the relative speeds of processes. We believe that it is important to free the programmer from timing information in a parallel computing environment. This is the motivating factor for our design decision to make the non-determinism insensitive to deadlocks.

Basic semantic ideas

Our study of the Angelic cc languages is based on viewing the minimal stores resulting from the execution of a process in an initial store as the fundamental observables. Informally, we model nondeterminism using the set of the minimal results obtainable along all the possible execution paths. The above example helps to illustrate these ideas. The possible resulting stores when the process is executed in the initial store with the unique constraint $(X = 1)$ are $(X = 1, Y = 2)$ and $(X = 1, Z = 3)$. Both these stores are minimal since the information contained in them is incomparable. This view of processes immediately discharges failing branches; since the inconsistent store entails any other store and is thus the maximal element in the lattice of stores, ordered by eqntailment.

Angelic non-derminism vs Demonic non-determinism

The kind of nondeterminism being discussed here is unrelated to demonic non-determinism. In demonic non-determinism (also called "indeterminism"), *one* of several paths will be pursued by the agent. However, in angelic nondeterminism as we discuss it, *all* branches will simultaneously be pursued by the agent: It should be kept in mind, however, that the environment can still influence which computation paths taken by the agent *terminate successfully*, through the imposition of constraints on shared variables. Note also that *synchronization* is still present in the system because it is possible to express that that an agent (e.g., $X = 1 \rightarrow Y = 2$) must block until some other agents have engaged in a particular action (imposed a constraint that entails $X = 1$ in the current store). Thus the work on the semantics of "committed-choice" indeterminacy for (concurrent) (constraint) logic programming languages [dBP90a,dBP90b,GCLS88,SRP91], is not directly relevant here. These papers are concerned with demonic non-determinism — and in that setting it is necessary to give a semantic account of deadlock and (proper) termination.

3 Angelic cc: operational semantics

The semantic domain of interest is parametrized by the *constraint system* underlying the given cc language. Briefly, a constraint system is an algebraic lattice [2], with some additional structure (the *cylindrification* operations $\exists_X$, for X a variable, and *diagonal* elements $\mathbf{d}_{XY}$, for X, Y variables) to allow for information

[2] Essentially, a Scott information system with the consistency structure removed, since in the constraint setting, the inconsistent or overdefined state of affairs is quite conceivable

Syntax.

$$
\begin{array}{rl}
\textit{Programs} & P ::= D.A \\
\textit{Declarations} & D ::= \epsilon \mid p(X) :: A \mid D.D \\
\textit{Agents} & A ::= c \mid c \rightarrow A \mid A \wedge A \mid A \vee A \mid \exists X A \mid p(X)
\end{array}
$$

Transition System.

$$
\begin{array}{rl}
\textit{Configurations} & (\gamma \in)\Gamma ::= A \mid \exists X(d, \gamma) \\
\textit{Labels} & (\lambda \in)\Lambda = B(E) \times B(E) \\
\textit{Transition relation} & \longrightarrow \subseteq Env \times \Gamma \times \Lambda \times \Gamma
\end{array}
$$

Axioms.

$$c \stackrel{(d, c \sqcup d)}{\longrightarrow} \mathbf{true} \quad (c \neq \mathbf{true}) \qquad c \rightarrow A \stackrel{(d,d)}{\longrightarrow} A \quad \text{if} \quad d \geq c \tag{1}$$

$$\frac{A \stackrel{\lambda}{\longrightarrow} A'}{\begin{array}{c} A \wedge B \stackrel{\lambda}{\longrightarrow} A' \wedge B \\ B \wedge A \stackrel{\lambda}{\longrightarrow} B \wedge A' \end{array}} \qquad \frac{A \stackrel{\lambda}{\longrightarrow} A'}{\begin{array}{c} A \vee B \stackrel{\lambda}{\longrightarrow} A' \\ B \vee A \stackrel{\lambda}{\longrightarrow} A' \end{array}} \tag{2}$$

$$\frac{A \stackrel{(\exists_X c, d)}{\longrightarrow} B}{\exists X A \stackrel{(c, c \sqcup \exists_X d)}{\longrightarrow} \exists X(d, B)} \qquad \frac{A \stackrel{(d \sqcup \exists_X c, d')}{\longrightarrow} B}{\exists X(d, A) \stackrel{(c, c \sqcup \exists_X d')}{\longrightarrow} \exists X(d', B)} \tag{3}$$

$$\rho \vdash p(X) \stackrel{(d,d)}{\longrightarrow} \exists \alpha(\mathbf{d}_{\alpha X}, \rho(p)) \tag{4}$$

Above, c ranges over basic constraints, that is, finite sets of tokens. α is some variable in the underlying constraint system which is not allowed to occur in user programs. (It is used as a dummy variable during parameter passing.)

Table 1: Syntax and transition system for the Ask-and-Tell Non-determinate cc languages

hiding and parameter passing. Intuitively, the state of a computation as given by the store, is described by an element of such a constraint system. For more details, the reader is referred to [SRP91]. In the following, for any partial order E, we let $B(E)$ stand for the set of finite elements of E.

The set $(\gamma \in)\Gamma$ of *configurations* is just the set of agents, augmented to allow for "local" stores within the scope of a quantifier (see discussion below). The set $(\lambda \in)\Lambda$ of labels is just the set of pairs of finite constraints, corresponding to the store at the beginning and end of the transition. We define a quartic *transition* relation

$$\longrightarrow \subseteq Env \times \Gamma \times \Lambda \times \Gamma$$

which will be used to define the operational semantics of the programming language. (Here $(\rho \in)Env$ is the set of all partial functions from procedure names to (syntactic) agents; we let ρ_0 be the function with empty domain.) Rather than write $\langle \rho, \lambda, A, B \rangle \in \longrightarrow$ we shall write $\rho \vdash A \stackrel{\lambda}{\longrightarrow} B$ (omitting the "$\rho \vdash$" if it is not relevant) and take that to mean that when initiated in store c, agent A can upgrade the store to d, and subsequently behave like B. In the usual SOS style, this relation will be described by specifying a set of axioms, and taking the relation to be the smallest relation satisfying those axioms.

The syntax of, and transition system for, the non-determinate language are summarized in Table 1. (For purposes of exposition we assume that procedures take exactly one variable as a parameter and that no program calls an undefined procedure.) The axioms for ask and tell operations are fairly obvious. [3] The axiom for $A \wedge B$ reflects the fact that A and B never communicate synchronously in $A \wedge B$. Instead, all communication takes place asynchronously with information added by one agent stored in the shared constraint for the other agent to use. Disjunction is modelled by making an arbitrary choice of which branch to follow, and discarding the other branch. *The difference between angelic and demonic non-determinism will arise in the way observations are defined, rather than the way in which configurations and transition relations are defined: by definition of observations we will ensure that a failed computation branch cannot be "observed" unless all other branches also fail.* [4]

In order to define the transition relation for $\exists X A$, we extend the transition relation to agents of the form $\exists X(d, A)$, where d is an internal store holding information about X which is hidden outside $\exists X(d, A)$. The

[3] Throughout the rest of this paper, depending on context, we shall let c stand for either a syntactic object consisting of a finite set of tokens, the constraint obtained by taking the closure of that set under $\vdash$, the (semantic) process that imposes that constraint on the store, or the (syntactic) agent that imposes that constraint on the store.

[4] It is possible, of course, to give a more "faithful" operational semantics which represents disjunctive configurations explicitly. This is done, e.g., in [Sar89]: we have chosen the present route for simplicity.

transition axiom for $\exists X A$ yields an agent with an internal store, reflecting the fact that all information about X in c is hidden from $\exists X A$, and all information about X that A produces is hidden from the environment. Note that B may need the produced information about X to progress; this information is stored with B in the constraint d. The axiom for agents with an internal store is straightforward. The information from the external store is combined with information in the internal store, and any new constraint generated in the transition is retained in the internal store. Note that this gives a treatment for procedure calls and local variables that does not need to use any of the machinery for "renaming variables apart" necessary in conventional accounts of logic programming.

Finally, procedure calls are handled by looking up the procedure in the environment ρ.

3.1 Operational Semantics

In order to extract an environment from $D.A$ in which to run A, we define a function $\mathcal{R}$ from declarations to (syntactic) environments (maps from procedure names to agents):[5]

$$\begin{aligned}&\mathcal{R}(\epsilon)\rho = \rho\\ &\mathcal{R}(p(X) :: A.D)\rho = \mathcal{R}(D)\rho[p \mapsto (\exists X \mathbf{d}_{\alpha X} \wedge A)]\end{aligned}$$

A computation in this transition system is a sequence of transitions in which the environment is constrained to produce nothing. Hence the final constraint of each transition should match the initial constraint of the succeding transition:

Definition 3.1 A c-sequence s for a program $D.A$ is a possibly infinite sequence $\langle c_i, A_i\rangle_i$ of pairs of agents and stores such that $c_0 = c$ and $A_0 = A$ and for all i, $\rho \vdash A_i \xrightarrow{(c_i, c_{i+1})} A_{i+1}$, where $\rho = \mathcal{R}(D)\rho_0$. Such a transition sequence is said to *terminate in* d if it is finite, of length $n \geq 1$ and A_{n-1} is *stuck* in $d = c_{n-1}$ (that is, there is no constraint e and agent B such that $\rho \vdash A_{n-1} \xrightarrow{(c_{n-1}, e)} B$). □

At any stage of the computation there may be several enabled transitions, each of which reduces one of the agent's subagents. Note that if in a computation sequence an agent is enabled infinitely often, it must be enabled everywhere except at a finite number of points. We say that a c-sequence s is *fair* if it contains no subagent enabled almost everywhere.

Now, we are ready to define the observables. The basic intuition is to model finite observations. Informally, an observation is finite if it can be made in a finite amount of time. For example, an observation of form "$X = 1$" is deemed to be finite. The formal statement of this idea of finiteness is in terms of the recursive enumerability of the set of observables: the set of observables of the computation of a process with a initial starting store must be recursively enumerable. The set of observables that we present here forms a recursive enumerable set.

The situation is complicated by the presence of non-determinism. Consider the computation of a process P starting from an initial store. The result of a computation is a set of stores, say S. Motivated by total correctness considerations, we take the viewpoint that the minimal stores of S are the important constituent of S. Since we allow infinite computations in our framework, these minimal stores could be infinite and thus not finitely observable.

Following previous work in the semantics of process calculi [Hen88], we take the view that (finite) observations are finite disjunctions of finite stores:

$$OBS = \{o \mid o \subseteq_f B(E)\}$$

where E is the underlying constraint system, and $\subseteq_f$ is the "is-a-finite-subset-of" relation. A store c is said to *satisfy* an observation $o = \{d_0, \ldots, d_{n-1}\}$ (written $c \vdash o$ — note that we are overloading the $\vdash$ symbol here) if $c \geq d_j$, for some $j < n$. A program P starting in an initial store c satisfies such an observation o if the following holds. Given *any* valid (possibly infinite) computation of the program, one of the constraints d_i is exceeded at a finite point of the computation. The formal definition of $Obs(P)(c)$ is given below. The abstract semantics clarifies the relationship between $Obs(P)(c)$ and the minimal elements of the set of stores obtained by executing P with starting state c.

[5]For simplicity, we shall assume that a procedure is declared only once in a program. The term $\rho[p \mapsto A]$ indicates the syntactic environment identical to ρ except that at p it takes on the value A.

Definition 3.2 For any program P and constraint c, the *observations of P at c*, written $Obs(P)(c)$ is defined as follows. Let $\langle c_i, A_i\rangle_i$ be any fair computation sequence. Then,

$$Obs(P)(c) = \{o \in OBS \mid \forall \text{fair } c\text{-sequences } \langle c_i, A_i\rangle_i \text{ of } P.\exists j.\ c_j \vdash o\}$$

$\square$

Example 3.1 Note that the agents $(X = a \rightarrow Y = b) \vee (X = a \wedge Y = b)$ and $(X = a \rightarrow Y = b)$ are operationally identical. That is, for any store c,

$$Obs((X = a \rightarrow Y = b) \vee (X = 1 \wedge Y = b))(c) = Obs(X = a \rightarrow Y = b)(c)$$

This agrees, of course, with both the logical and denotational semantics. In intuitionistic logic, both the sequents $X = a \rightarrow Y = b \vdash (X = a \rightarrow Y = b) \vee (X = a \wedge Y = b)$ and $(X = a \rightarrow Y = b) \vee (X = a \wedge Y = b) \vdash X = a \rightarrow Y = b$ are provable. From the denotational point of view, we shall see that the set of minimal results produced by both agents in any store will also coincide. $\square$

The following points should be emphasized:

Handling failing computations: A failed computation results in an inconsistent store. Note that an inconsistent store s satisfies all observations: `false` $\vdash o$, for all $o \in OBS$. In other words, failing computation paths do not affect the observables of a process. Furthermore, note that the observation $\{$`false`$\}$ is satisfied only if *all* computation paths end in failure.

Deadlock: Note that only failed and non-failed computations can be distinguished. There are no mechanisms to distinguish deadlocked and non-deadlocked computations. In our setting, this seems a reasonable assumption to make since the item of interest is only the state of the store.

Total correctness: The observables have the flavor of total correctness reasoning. This is because we demand that every valid computation sequence satisfies the logical assertions given by the observables.

4 Angelic cc: Denotational semanics

In determinate computations, the state of the system can be described by a single constraint, corresponding to the current store. However, in a non-determinate computation, the store may have split into several stores, and so the state of the system must be modelled by a set of constraints. It becomes necessary therefore, to represent a "free" *disjunction* of constraints. Given a (cylindric) constraint system E over a token-set D, the most straightforward way to achieve this is to define a new constraint system E' with tokens being finite subsets of D, and an entailment relation derived from that of D which treats these tokens disjunctively:

$$\{u_0, \ldots, u_{n-1}\} \vdash_{E'} \{x_0, \ldots, x_{k-1}\}$$

iff for every *choice set* s for $\{u_0, \ldots, u_{n-1}\}$ (that is, every set s with a non-empty intersection with each u_i) there is a $j < k$ such that $s \vdash_E x_j$. But this, of course, is just the construction of the Smyth power-domain over an information system [Sco82]! Therefore, in order to define the state of the computation, we shall make use of $\mathcal{P}_I(E)$, where E is the underlying constraint system. Further, this construction comes equipped with a "singleton" operator $:\{\ \_\ \}: : E \rightarrow \mathcal{P}_I(E)$ (which maps an element $d \in E$ to the element $:\{\ d\ \}: =\uparrow e$ of $\mathcal{P}_I(E)$) and a non-deterministic union operation $\uplus : \mathcal{P}_I(E) \times \mathcal{P}_I(E) \rightarrow \mathcal{P}_I(E)$ (which, in this representation is just set-union). We shall denote the ordering on this power-domain by $\preceq$.

The representation theorem for the Smyth powerdomain(see, e.g., [Plo82, Section 8]) gives a simpler characterisation of the elements of $\mathcal{P}_I(E)$. In particular, $\mathcal{P}_I(E)$ is isomorphic to the domain $Rep(\mathcal{P}_I(E))$ of up-closed Scott-compact[6] subsets of E ordered by inverse set-inclusion . Furthermore, since E is an algebraic lattice, an up-closed Scott compact subset S of E is completely determined by the set of minimal elements of S. Informally, the representation theorem is the tool that relates the two differing operational viewpoints: the view of "finite disjunction of finite constraints"(which corresponds to $\mathcal{P}_I(E)$) and the view of "minimal elements of a set of resulting stores"(which corresponds to $Rep(\mathcal{P}_I(E))$).

[6] A subset S of a partial order D is said to be Scott-compact if any cover of S (that is, a set T such that for all $x \in S$ there is a $y \in T$ with $y \leq x$) consisting of finite elements of D contains a finite subset that is also a cover. Such sets are compact in the Scott topology and, intuitively, correspond to the "finitely generable" sets of elements of D.

We turn now to the question: What should the denotation of a process be? The most apparent answer is to treat it as a function in $E \rightarrow \mathcal{P}_I(E)$, which associates with each input store the disjunction of the limits of the fair execution sequences of the program. However, it is much more convenient to treat it instead as a *linear* function in $\mathcal{P}_I(E) \rightarrow \mathcal{P}_I(E)$: the answer returned is just the disjunction of the answers returned on invoking the program on each of the disjuncts of the input; this corresponds to the operational intuition that the possible results of a program invoked in one of a possible set of stores is obtained by collecting the answers obtained by executing the program in each of the input stores. For any continuous function $f : E \rightarrow \mathcal{P}_I(E)$ we shall let $f^\dagger$ indicate its unique linear extension.But what sort of function is the denotation of a process?

Arguments similar to [JPP89,SRP91] can be made here. On any input, a program will only produce an output with more information (this is a fundamental property of the ask and tell primitives); hence the function will be *extensive*. Also, a disjunct in the answer is obtained by taking the limits of fair execution sequence—hence the function will be idempotent. And, of course, since we are interested in an implementable programming language, the functions will be monotone and continuous. Thus, the denotations of programs will be *continuous linear closure operators* on $\mathcal{P}_I(E)$. The set of such operators over E is designated $\mathcal{CP}_I E$ and ordered extensionally: $f \leq g$ iff for all $x \in \mathcal{P}_I(E)$, $f(x) \leq g(x)$.

As is well-known, a closure operator can be represented by the set of its fixed-points, and this property was used to present a very simple semantics for the determinate cc programming languages in [SRP91]. In the present case, this would mean that a process can be represented by a family of sets of constraints. However, we now show that it is possible to represent a process as just a set of constraints.

Definition 4.1 A *singleton fixed-point* (also called *resting-point*) of a function $f : \mathcal{P}_I(E) \rightarrow \mathcal{P}_I(E)$ is an element $d \in E$ such that $f(:\{ d \}:) = :\{ d \}:$. □

We show that f can be recovered from the set S_f of its singleton fixed-points. For any x, $f(x)$ has a complete set of minimal elements[7], since it is a Scott-compact subset of a complete algebraic lattice. Further, every minimal element d of $f(x)$ is in S_f: for, $f(x) = f(f(x))$ implies that $d \in f(:\{ e \}:)$ for some $e \in f(x)$ (f is linear), and since d is minimal in $f(x)$, it must be the case that $d \leq e$. But by extensiveness, $e \leq d$; hence $d = e$, and we can recover the effect of f on x from the formula $f(x) = \uparrow (x \cap S)$.

In the other direction, assume S is Scott-compact. Then, for all $x \in E$, so is $\uparrow x \cap S$ since for complete algebraic lattices Scott-compact sets are closed under intersection (and $\uparrow x$ is easily seen to be Scott-compact). It is not hard to see that the linear extension of $f = \lambda x. \uparrow (\uparrow x \cap S)$ defines a closure operator on $\mathcal{P}_I(E)$. Thus we can state:

Theorem 4.1 *Let E be an algebraic lattice. $S \subseteq E$ is the set of singleton fixed-points of a linear closure operator f on $\mathcal{P}_I(E)$ iff it is Scott-compact. Furthermore, f is the linear extension of $\lambda x. \uparrow x \cap S$.*

In addition, f is continuous iff for all chains $\langle c_i \rangle_i$, $S \cap \uparrow \sqcup c_i = \bigcap_i (S \cap \uparrow c_i)$.

This representation of $f \in \mathcal{CP}_I(E)$ is so convenient that from now on we shall not distinguish between f and S_f.

For E an algebraic lattice (with finite top), continuous linear closure operators over $\mathcal{P}_I(E)$ form an algebraic, distributive lattice, with joins given by intersection of the set of singleton fixed-points, and meets given by union. The associated ordering $\leq$ on operators is just the extensional ordering on functions.

Definition of combinators

We now define the combinators.

The process c augments its input with (the finite constraint) c. Thus it behaves as the operator $(\lambda x.:\{ c \}: \sqcup x)^\dagger$, which in terms of resting-points, is just:[8]

$$c = \{d \in E \mid d \geq c\}$$

Let c be a constraint, and f a process. The process $c \rightarrow f$ waits until the store contains at least as much information as c. It then behaves like f. Such a process can be described by the function $(\lambda x. \mathit{if}\ x \geq :\{ c \}: \mathit{then} f(x)\ \mathit{else}\ :\{ x \}:)^\dagger$. In terms of its range:

$$c \rightarrow f = \{d \in E \mid d \geq c \Rightarrow d \in f\}$$

[7] that is, every $y \in f(x)$ lies above a minimal element of $f(x)$

[8] Recall that we are representing linear closure operators over $\mathcal{P}_I(E)$ by the set of their singleton fixed-points (i.e., resting-points).

Semantic Equations.

$$\mathcal{N}(c)e = \{d \mid d \geq c\}$$
$$\mathcal{N}(c \rightarrow A)e = \{d \mid d \geq c \Rightarrow d \in \mathcal{N}(A)e\}$$
$$\mathcal{N}(A \wedge B)e = \{d \mid d \in \mathcal{N}(A)e \wedge d \in \mathcal{N}(B)e\}$$
$$\mathcal{N}(A \vee B)e = \{d \mid d \in \mathcal{N}(A)e \vee d \in \mathcal{N}(B)e\}$$
$$\mathcal{N}(\exists X A)e = \{d \mid \exists c \in \mathcal{N}(A)e.\exists_X d = \exists_X c\}$$
$$\mathcal{N}(p(X))e = \exists_\alpha(d_{\alpha X} \sqcup e(p))$$
$$DD(\epsilon)e = e$$
$$DD(p(X) :: A.D) = DD(D)e[p \mapsto \exists_X(d_{\alpha X} \sqcup \mathcal{N}(A)e)]$$
$$\mathcal{N}(D.A) = \mathcal{N}(A)(\ \underline{\texttt{fix}}\ DD(D))$$

Above, c ranges over basic constraints, that is, finite sets of tokens. α is some variable in the underlying constraint system which is not allowed to occur in user programs. (It is used as a dummy variable during parameter passing.) e maps procedure names to processes, providing an environment for interpreting procedure calls. We use the notation $c \sqcup f$ to stand for $\{c \sqcup d \mid d \in f\}$.

Table 2: Denotational semantics for the Angelic cc languages

The ask operation is monotone and continuous in its process argument.

Consider the parallel composition of two processes f and g. Suppose on input c, f runs first, producing $f(c)$. Because it is idempotent, f will be unable to produce any further information. However, g may now run, producing some more information, and enabling additional information production from f. The system will quiesce exactly when *both* f and g quiesce. Therefore, the set of resting-points of $f \wedge g$ is exactly the intersection of the resting-points of f and g:

$$f \wedge g = f \cap g$$

Clearly, this operation is well-defined, and monotone and continuous in both its arguments. The simplicity of this definition of parallel composition is one of the attractions of our approach. For instance, the commutativity, associativity and idempotence of parallel composition are immediate from the definition, and all arguments about iterating to a limit are completely avoided.

Consider the non-deterministic composition of two processes f and g. A resting-point for $f \vee g$ must be a resting-point of either f or g:

$$f \vee g = f \cup g$$

As above, this operation is well-defined, and monotone and continuous in both its arguments. Clearly from the definition, disjunction is commutative, associative and idempotent, and distributes (both ways) with parallel composition (conjunction).

We turn finally to projection. Suppose given a process f. We wish to define the behavior of $\exists X f$, which, intuitively, must hide all interactions on X from its environment. Consider the behavior of $\exists X f$ on input c. c may constrain X; however this X is the "external" X which the process f must not see. Hence, to obtain the behavior of $\exists X f$ on c, we should observe the behavior of f on $\exists_X c$. However, $f(\exists_X c)$ may constrain X, and this X is the "internal" X. Therefore, the result seen by the environment must be $c \sqcup \exists_X f(\exists_X c)$. This leads us to define:

$$\exists X f = \{c \in E \mid \exists d \in f.\exists_X c = \exists_X d\}$$

These hiding operators enjoy several interesting properties. For example, we can show that they are "dual" closure operators (i.e., kernel operators), and also cylindrification operators on the class of denotations of non-determinate programs.

Recursion is handled in the usual way, by taking limits of the denotations of all syntactic approximants, since the underlying domain is a cpo and all the combinators are continuous in their process arguments. The resulting semantic function $\mathcal{N} : Agents \rightarrow \mathcal{CP}_I(E)$ utlilizing these operators is given by Table 2.

4.1 Relationship between the two semantics

We informally discuss the key idea underlying the proof of coincidence of the operational and denotational semantics. The proof is an extension of proofs for the determinate case[JPP89,SRP91], to a non-determinate setting. This extension involves exploiting the algebraic properties of the Smyth powerdomain [Smy78], especially the properties relating to finite-branching.

Note that elements of $Obs(P)(c)$ can be viewed as elements of $\mathcal{P}_I(E)$. Thus, the following definition makes sense.

Definition 4.2 $\mathcal{O}_\infty(P)(c) = \sqcup\{o | o \in Obs(P)(c)\}$. □

$\mathcal{O}_\infty(P)$ is extended to the whole of $\mathcal{P}_I(E)$ by extending it linearly and continuously.

The following lemma captures formally the intuition that enabled transitions are not disabled by addition of information to the store.

Lemma 4.2 (Operational monotonicity for angelic cc.) *If $A_1 \xrightarrow{(c,d)} A_2$ is a possible transition and $e \geq c$ then $A_1 \xrightarrow{(e,d\sqcup e)} A_2$.*

Proof 4.2 Proof is a simple structural induction on agents. □

Lemma 4.3 *[Relating $\mathcal{O}_\infty$ to reduction:]*

1. *Let $A_1 \xrightarrow{(c,d)} A_2$ be a possible transition. Then, $\mathcal{O}_\infty(A_1)(c) \leq \mathcal{O}_\infty(A_2)(c)$.*
2. *Let $A \xrightarrow{(c,d_i)} A_i, i = 1 \dots n$ be all possible transitions from A with initial state c. Then, $\mathcal{O}_\infty(A)(c) = \uplus_i \mathcal{O}_\infty(A_i)(c_i)$.*

Proof 4.3

1. From definitions, every element of $Obs(A_1)(c)$ is also an element of $Obs(A_2)(d)$: as, for any P, c, $o \in Obs(P)(c)$ implies that every disjunct containing o is also in $Obs(P)(c)$.
2. Follows by noting that
$$Obs(A)(c) = \{o_1 \vee o_2 \dots o_n | o_i \in Obs(A_i)(c_i)\}$$

□

Theorem 4.4 *$\mathcal{O}_\infty(P)$, the continuous extension of $Obs(P)$ is a closure operator on $|D|$.*

Proof 4.4 Monotonicity, continuity and the fact that it is increasing follow from the definitions.

Since, $\mathcal{O}_\infty(P)(c) = \sqcup\{o | o \in Obs(P)(c)\}$, it suffices to show that $\mathcal{O}_\infty(P)(o) \leq \mathcal{O}_\infty(P)(c)$, for all $o \in Obs(P)(c)$. Let $o = \{c_1, \dots, c_n\}$. Then, $\mathcal{O}_\infty(P)(o) = \uplus_i \mathcal{O}_\infty(P)(c_i)$. Without loss of generality, assume the following:

- $c \leq c_i$, for $i = 1 \dots n$.
- Every subset of o is strictly greater than o in the order relation.

In terms of observations, the second restriction amounts to saying : pick a disjunct c_i; then, there is some c-sequence of P such that c_i is exceeded at a finite stage of computation.

Consider the tree of all c-sequences. Since $o \in Obs(P)(c)$, any c-sequence exceeds one of the conjuncts at a finite stage. Since the tree of c-sequences is finitely branching, from Konig's lemma, there is a finite subtree such that the stores in each of the leaves of the finite subtree entail the disjunctive constraint o. Let the finite leaves be $(S_1, P_1) \dots (S_m, P_m)$. Each of these stores satisfies the disjunct o: thus each can be labelled with a constraint c_i that it exceeds. Furthermore, from the second condition on o, for each c_i, there is at least one store S_j that exceeds it. Construct a partition $J_1 \dots J_m$ of $\{(S_1, P_1) \dots (S_m, P_m\}$, such that the stores in J_i exceed c_i. Then, from operational monotocity, there is a reduction path from (P, c_i) to (P', c'), for every $(P', c') \in J_i$. Using first part of lemma 4.3,

$$\mathcal{O}_\infty(P)(o) \leq \uplus_i \mathcal{O}_\infty(P')(c')$$

However, from the second part of lemma 4.3,

$$\uplus_i \mathcal{O}_\infty(P')(c') = \mathcal{O}_\infty(P)(c)$$

Thus, $\mathcal{O}_\infty(P)(o) \leq \mathcal{O}_\infty(P)(c)$. Hence, the result. □

Theorem 4.5 $\mathcal{O}_\infty(P) = \mathcal{N}(P)$.

Proof 4.5 The proof proceeds by structural induction on P.

c: The base case is for programs of form d. From the operational semantics, $\mathcal{O}_\infty(c)(d) = c \sqcup d$. Result follows as $c \sqcup d$ is the least element above c and d.

$c \rightarrow A$: From the operational semantics,

$$\mathcal{O}_\infty(c \rightarrow A)(d) = \begin{cases} d, \text{ if } c \not\leq d \\ \mathcal{O}_\infty(A)(d), \text{ otherwise} \end{cases}$$

¿From inductive hypothesis, $\mathcal{O}_\infty(A)(d) = \mathcal{N}(A)(d)$. Hence, the result.

$A_1 \wedge A_2$: For this case, we prove first that $\mathcal{O}_\infty(A_1 \wedge A_2)$ is the smallest closure operator above $\mathcal{O}_\infty(A_1)$ and $\mathcal{O}_\infty(A_2)$. Note that result follows immediately: From inductive hypothesis, $\mathcal{O}_\infty(A_1) = \mathcal{N}(A_1)$ and $\mathcal{O}_\infty(A_2) = \mathcal{N}(A_2)$). Thus, $\mathcal{O}_\infty(A_1) \sqcup \mathcal{O}_\infty(A_2) = \mathcal{N}(A_1) \sqcup \mathcal{N}(A_2)$).

To prove that $\mathcal{O}_\infty(A_1) \sqcup \mathcal{O}_\infty(A_2) = \mathcal{O}_\infty(A_1 \wedge A_2)$, we show that the singleton fixpoint sets of the two closure operators match. It is immediate from definitions that $\mathcal{O}_\infty(A_1) \sqcup \mathcal{O}_\infty(A_2) \leq \mathcal{O}_\infty(A_1 \wedge A_2)$. For the converse $\mathcal{O}_\infty(A_1 \wedge A_2) \leq \mathcal{O}_\infty(A_1) \sqcup \mathcal{O}_\infty(A_2)$, note that if $\mathcal{O}_\infty(A_1)(d) = \mathcal{O}_\infty(A_2)(d) = d$, then $\mathcal{O}_\infty(A_1 \wedge A_2)(d) = d$, as neither A_1 nor A_2 add any information to d.

$A_1 \vee A_2$: For this case, note that from lemma 4.3 and operational monotonicity,

$$\mathcal{O}_\infty(A_1 \vee A_2) = \mathcal{O}_\infty(A_1) \uplus \mathcal{O}_\infty(A_2)$$

Result follows from induction hypothesis.

$\exists X A$: From the operational semantics, $\mathcal{O}_\infty(\exists X A)(c) = d$ implies that there is an internal store d' such that $\mathcal{O}_\infty(A)(\exists_X c) = d'$ and $\exists_X d' \sqcup c = d$. From inductive hypothesis, $d' = \mathcal{N}(A)(\exists_X c)$. From definitions, $\exists_X d' \sqcup c = \mathcal{N}(\exists X A)(c)$.

Procedures: This case of structural induction corresponds to procedure calls $p(X)$. For simplicity of notation, we will assume that the procedures are not mutually recursive. Let the procedure definition be $p(X) :: A$. Note that

$$\begin{aligned} \mathcal{O}_\infty(p(Y)) &= \sqcup_k[\mathcal{O}_\infty(p^k(Y))] \\ \mathcal{N}(p(Y)) &= \sqcup_k[\mathcal{N}(p^k(Y))] \end{aligned}$$

where p^k is the program that corresponds to k unwindings of the procedure: more formally, it is defined by induction on k, as follows:

$$\begin{aligned} p^0(Y) &:: \texttt{true} \\ p^{k+1}(Y) &:: A[p/p^k] \end{aligned}$$

We prove by induction on k that $\mathcal{O}_\infty(p^k(Y)) = \mathcal{N}(p^k(Y))$. The base case for $k = 0$ is true since both sides equal the identity closure operator. Assume the result for $k = m$. Consider $k = m + 1$. Then, from definitions,

$$\begin{aligned} \mathcal{O}_\infty(p^{m+1}(Y))(c) &= \mathcal{O}_\infty(\exists_\alpha(d_{\alpha Y}, A[p/p^m]) \\ \mathcal{N}(p^{m+1}(Y)) &= \exists_\alpha(d_{\alpha Y} \sqcup \exists_X[d_{\alpha,X} \sqcup \mathcal{N}(A[p/p^k])] \end{aligned}$$

Note that $\exists_X[d_{\alpha,X} \sqcup \mathcal{N}(A[p/p^k])]$ is equal to $\mathcal{N}(A[p/p^k, X/\alpha])$. Now, using the inductive hypothesis on k and the structural induction hypothesis, the two closure operators above are equal. Hence, the result.

□

Theorem 4.6 (Full abstraction) *$\mathcal{N}(P) = \mathcal{N}(Q)$ iff for all contexts $\mathcal{C}[\bullet]$, $Obs(\mathcal{C}[P]) = Obs(\mathcal{C}[Q])$.*

Proof 4.6 Proof is a straightforward application of the above lemmas and the compositionality of the denotational semantics.

$$\begin{aligned} &\mathcal{N}(P) = \mathcal{N}(Q) \\ &\quad \leftrightarrow \quad \mathcal{N}(\mathcal{C}[P]) = \mathcal{N}(\mathcal{C}[Q]) \quad \text{(Compositionality of } \mathcal{N}) \\ &\quad \leftrightarrow \quad \mathcal{O}_\infty(\mathcal{C}[P]) = \mathcal{O}_\infty(\mathcal{C}[Q]) \quad \text{(Theorem 4.5)} \\ &\quad \leftrightarrow \quad Obs(\mathcal{C}[P]) = Obs(\mathcal{C}[Q]) \end{aligned}$$

□

5 Success set semantics

In this section, we sketch the semantic treatment for a different notion of observation, that equates divergence and failure and records the "guarantees" that hold for finite and terminating computations only. We assume that the constraint system is such that the entailment relation is finitary, i.e there is no infinite sequence of constraints c_i, such that c_i entails c_{i+1} and $c_i \neq c_{i+1}$.[9] Thus, there are no infinite descending chains of finite elements.

As before, we first define the notion of observables. Informally, the main difference from the previous definition is that we record only the results of terminating computations and disregard non-terminating computations. More formally the notion of observable is defined as follows. Let $Min(S)$ denote the set of minimal elements of S. (The finitariness assumption guarantees the existence of a complete minimal set of elements)

Definition 5.1 For any program P and constraint c, the *observations of P at c*, written $Obs(P)(c)$ is given by:

$$Obs(P)(c) = Min(\{d \mid \text{ there is a } c\text{-sequence terminating in } d\})$$

$\square$

We now turn to a discussion of the semantic model for this notion of observation. Given a finitary constraint system $(D, \sqsubseteq)$, let $P(D)$ denote the constraint system defined as follows: The elements are non-empty subsets of D. The entailment preorder is defined as

$$S_1 \sqsubseteq S_2 \;\Rightarrow\; (\forall y \in S_2)\ (\exists x \in S_1)\ [x \sqsubseteq y]$$

Note the difference from the disjunctive constraint systems. Here we have arbitrary subsets of D as opposed to the finite subsets of D that were used to generate disjunctive constraint systems. $P(D)$ can be turned into a partial order by quotienting by the equivalence of the preorder. Indeed, the partial order so obtained is isomorphic to the partial order of upwards closed subsets of D, ordered under reverse inclusion. Call this partial order $\mathcal{P}_J(D)$. The elements of $\mathcal{P}_J(D)$ are determined uniquely by the subset of the minimal elements.

$\mathcal{P}_J(D)$ is closed under greatest lower bound of descending chains—the glb of a descending chain $\langle S_i \rangle_i$ is $\cup_i S_i$. The finitary assumption on the entailment relation allows us to deduce that $\cup_i S_i$ has a complete set of minimal elements.

The semantic functions are linear closure operators on the constraint system $\mathcal{P}_J(D)$, i.e closure operators that satisfy $f(S) = \cup\{f(s) \mid s \in S\}$. Note the absence of continuity conditions. The point is that in the absence of infinite computations, it suffices to consider the action of the semantic function on finite elements. As before, the linear closure operators are characterized by their singleton fixed-point sets. The assumption on existence of minimal elements in the sets constituting $\mathcal{P}_J(D)$ replaces the use of Scott-compactness in the original proof to deduce the existence of minimal elements.

Furthermore, the linear closure operators are ordered by inclusion of resting-points. In other words, the ordering is the *reverse* extensional ordering. The least upper bound of a chain $\langle f_i \rangle_i$ is defined pointwise by $f = \lambda x.\cup_i f_i(x)$. It is immediate that f is linear, the lub of f_i, and a closure operator. As before, the resting-points of f are the union of the resting-points of f_i.

The form of equations defining the combinators remains unchanged and it can be easily checked that the combinators are continuous functions on processes. For example, $f \wedge g = f \cap g$. Let $\langle f_i \rangle_i$ be chain with least upper bound f. Then, $f = \cup_i f_i$, and $f \wedge g = (\cup_i f_i) \cap g = \cup_i (f_i \cap g)$.

As before, the denotational semantics matches the operational semantics accurately. This proof follows the sketch of the analagous proof presented in the previous section and is omitted.

Theorem 5.1 (Full abstraction) $Obs(P)(c) = Min(\mathcal{N}(P)c)$

6 Conclusion

The main contribution of this paper is to extend the traditional logic programming view of programs with AND-parallelism as conjunction and OR-parallelism as disjunction to Angelic cc languages with ask, tell, parallel composition, hiding, recursion and angelic non-determinism ("parallel backtracking"). The results

[9] This assumption is actually not needed, and we shall remove it in the final version of the paper. We retain it here to keep the definitions simple.

of this paper are achieved by using domain theoretic tools to model constraints and search. The semantics here smoothly extends semantics for (constraint) logic programs, and determinate cc programs. The denotational semantics is shown to be fully abstract with respect to an operational semantics that observes (finite approximations to) limits of execution sequences. We have also presented another slightly different model which is fully abstract with respect to the minimal constraints produced along terminated execution paths. In both cases, the semantics cuts down to the (fully abstract) semantics for the special cases of (constraint) logic programs (no asks) and determinate cc programs (no disjunction).

The Ask-and-Tell cc languages — including the Angelic cc languages discussed in this paper — have been characterized in simple logical terms [LS91], thus providing additional evidence for the simplicity and naturalness of the notion of angelic non-determinism discussed here. In particular, it is shown that a cc agent A passes the test $c_1 \vee \ldots \vee c_n$ iff $A \vdash (c_1 \vee \ldots \vee c_n)$ in intuitionistic logic. Taken together, the results of the current paper and [LS91] establish a harmonious operational, logical and denotational characterization of the Angelic cc languages in which the proof-theory of a fragment of intuitionistic logic is related to the operational semantics of the cc languages, and the model-theory is related to the denotational semantics of the cc languages.

Acknowledgements. We gratefully acknowledge discussions with Prakash Panangaden and Keshav Pingali and Pat Lincoln.

References

[dBP90a] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint logic languages. In *Proceedings of CONCUR '90*, 1990.

[dBP90b] Frank de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. June 4 1990.

[DvH+88] M. Dincbas, P. van Hentenryck, H. Simonis F. Berthier. The constraint logic programming language chip. In *Proceedings of the FGCS Conference*, November 1988.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. Technical Report TR 32/89, Dipartimento di informatica, Universita di pisa, September 1989.

[GCLS88] Rob Gerth, Mike Codish, Yossi Lichtenstein, and Ehud Shapiro. A fully abstract denotational semantics for Flat Concurrent Prolog. In *LICS 88*, 1988.

[GS89] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proceedings of Sixteenth ACM Symposium on Principles of Programming Languages*, January 1989.

[Hen88] Matthew Hennessy. *Algebraic theory of processes.* MIT Press, 1988.

[HS91] Pascal Van Hentenryck and Vijay A. Saraswat. cc(FD): Constraint programming over finite domains. Technical report, Computer Science Department, Brown University, forthcoming 1991.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, January 1987.

[JPP89] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully-abstract semantics for a functional language with logic variables. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, pages 294–303, 1989.

[LS91] Patrick Lincoln and Vijay Saraswat. Proofs as Concurrent Processes: A Logical Interpretation for Concurrent Constraint Programming. Technical report, Systems Sciences Laboratory, Xerox PARC, November 1991.

[Mah87] Michael Maher. Logic semantics for a class of committed-choice programs. In *4th International Conference on Logic Programming*. MIT Press, May 1987.

[Plo82] G.D. Plotkin. Pisa lecture notes. Unpublished notes, 1981-82.

[Sar87a] Vijay A. Saraswat. The concurrent logic programming language **cp**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[Sar87b] Vijay A. Saraswat. **CP** as a general-purpose constraint language. In *Proceedings of the National Conference, Seattle.* American Association for Artificial Intelligence, July 1987.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[Smy78] M. B. Smyth. Powerdomains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[SRH91] Vijay A. Saraswat, Fransesca Rossi, and Pascal Van Hentenryck. Towards a general framework for constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, February 1991.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.