Truly Concurrent Constraint Programming

Vineet Gupta [*] Radha Jagadeesan [†] Vijay Saraswat [*]

Abstract

Concurrent Constraint Programming (CCP) is a powerful computation model for concurrency obtained by internalizing the notion of computation via deduction over (first-order) systems of partial information (constraints). In [SRP91] a semantics for indeterminate CCP was given via sets of bounded trace operators; this was shown to be fully abstract with respect to observing all possible quiescent stores (= final states) of the computation. Bounded trace operators constitute a certain class of (finitary) "invertible" closure operators over a downward closed sublattice. They can be thought of as generated via the grammar: $t ::= c \mid c \to t \mid c \wedge t$ where c ranges over constraints, $\wedge$ is conjunction and $\to$ intuitionistc implication.

We motivate why it is interesting to consider as observable a "causality" relation on the store: what is observed is not just the conjunction of constraints deposited in the store, but also the causal dependencies between these constraints — what constraints were required to be present in the computation before others could be generated. We show that the same construction used to give the "interleaving" semantics in [SRP91] can be used to give a true-concurrency semantics provided that denotations are taken to be sets of bounded closure operators, which can be generated via the grammar:

$$k ::= c \mid c \to k \mid k \wedge k$$

Thus we obtain a denotational semantics for CCP fully-abstract with respect to observing this "causality" relation on constraints. This semantics differs from the earlier semantics in preserving more fine-grained structure of the computation; in particular the Interleaving Law

$$(c \to A) \parallel (d \to B) = (c \to (A \parallel (d \to B))) \,\square\, (d \to (B \parallel (c \to A))) \tag{1}$$

is not verified ($\square$ is indeterminate choice). Relationships between such a denotational approach to true concurrency and different powerdomain constructions are explored.

1 Introduction

Concurrent constraint programming [Sar89,SR90,SRP91] is a simple and powerful model of concurrent computation obtained by internalizing the notion of computation as deduction

[*]Xerox PARC, 3333 Coyote Hill Road, Palo Alto Ca 94304; {vgupta,saraswat}@parc.xerox.com

[†]Dept. of Mathematical Sciences,Loyola University-Lake Shore Campus, Chicago, Il 60626; radha@math.luc.edu

over (first-order) systems of partial information. The model is characterized by monotonic accumulation of information in a distributed context: multiple agents work together to produce *constraints* on shared variables. A constraint (over a given finite set of variables) is a finitary specification of possibly partial information about the values the variable can take. A typical example of a constraint is a first order formula over some algebraic structure. Constraints come naturally equipped with an entailment relation: $c_1, \ldots, c_n \vdash c$ holds exactly if the presence of constraints $c_1, \ldots, c_n$ implies the presence of the constraint c. Thus constraints can combine additively, without any prejudice about their source or origin, to produce other constraints. An *agent* has access to a finite set of variables — the basic operations it may perform are to post constraints on some subset of variables it has access to (A::=c), to check whether a constraint is entailed by ones that have already been posted and if so, reduce to another agent, perhaps indeterminately (A::= $\Box_{i \in I} c_i \rightarrow A_i$), to create brand new variables (A::=$\exists X.A$), or to reduce to a parallel composition of other agents (A::=$A_1 \parallel A_2$).

Several authors have investigated the semantic framework of CCP languages [SRP91, dBP91]. It is natural to observe for every agent the final store obtained on executing the agent to quiescence (i.e., the final store). To obtain a compositional analysis, one needs to investigate the nature of interactions across a boundary between a system S and its environment E. (Both S and E should be thought of as consisting of a parallel composition of agents.) Typically, S and E will share some variables V. One may think of S as detecting the presence of some constraints c_1 (on V), producing constraints c_2, and then suspending until more constraints c_3 are produced, and then producing constraints c_4, and so on, until finally it produces a constraint c_{2n} and quiesces (that is, reaches a stage in which it is inert unless the environment provides some input). Such interactions may be described by means of the grammar

$$t ::= c \mid c \rightarrow t \mid c \wedge t$$

Each t is called a *trace*; the conjunction of all constraints appearing in t, denoted $|t|$, is called its *bound*. Mathematically, each such t can be taken to describe a certain class of finitary "invertible" closure operators over the lattice generated by the constraint systems, called *bounded trace operators* (bto's). (Recall that a closure operator over a lattice is an operator that is monotone, idempotent and increasing.) The denotation of an agent may then be taken to be the set of all btos that an agent can engage in. [SRP91] shows that program combinators can be defined over such a structure, and in fact such a denotational semantics is fully abstract with respect to observing final stores. Furthermore, it turns out that the semantics of the determinate fragment of CCP can in fact be described by a single closure operator, equivalent to the parallel composition of each of the btos. In what follows, we will call this the "standard model" of CCP.

It is important to point out that the nature of communication in CCP is somewhat different from that in other models of concurrency, such as actors, CCS, CSP, or imperative concurrency. In particular, the lack of "atomicity" of basic actions is already built into this model of CCP[1]. For instance, if a constraint c is logically equivalent to the conjunction of

[1]For the purposes of this paper, CCP refers to "eventual tell" version of CCP [Sar89], rather than the "atomic tell" version for which a similar claim cannot be made.

constraints a and b, the standard model validates the *Law of Non-Atomicity of Tells*

$$a \parallel b = c \tag{2}$$

and the *Law of Non-Atomicity of Asks*

$$a \rightarrow b \rightarrow A = c \rightarrow A \tag{3}$$

1.1 Causal semantics for CCP

Such a development of the semantics of CCP is not fully satisfactory for several reasons.

Interleaving of concurrent actions. The standard model is said to be an "interleaving" model because it validates the Interleaving Law (Law 1), which expresses that a parallel composition of agents can be reduced to a choice between all possible interleavings of its basic actions. In some sense, such a law indicates that the system is operating in "global time", or has a single notion of observer. This seems particularly unnatural in the context of CCP because of the asynchronous, distributed nature of the constraint store, and the notion of communication via information accumulation. There is no need ever for any "system-wide" global-time, or synchronization in an implementation of CCP; multiple agents may detect the presence of constraints in different order; it is therefore surprising that the semantic treatment reflects such a "global" time.

Further, the conflation of parallelism with non-deterministic choice means that is not possible for CCP compilers to transform input programs soundly using the laws valid in this model, while preserving the "degree of parallelism" of the input program. A natural question that arises then, is the formalization of the notion of "degree of parallelism" and the elaboration of a model which respects this notion. Clearly, such a model would have to be more fine-grained than (make more distinctions than) the standard model because it would have to invalidate the Interleaving Law. However, there are laws satisfied by the standard model which allow for useful compile time optimizations that respect degree of parallelism, e.g. the Law of Immediate Discharge:

$$a \parallel (a \rightarrow B) = a \parallel B \tag{4}$$

or the Law of Intermediate Causation:

$$\exists X.[b \rightarrow (X \parallel A) \parallel X \rightarrow B] = b \rightarrow (A \parallel B) \tag{5}$$

(where X is not free in A, B). These laws should be preserved by the new model.

More speculatively, we also aim to provide some semantic support to a question faced by a compiler designer: in what order should a compiler carry out a given set of optimizations? Intuitively, an optimization is applicable if and only if an associated precondition is satisfied (e.g., a variable can be put in a register if certain non-aliasing conditions are met); and once an optimization is carried out, certain post-conditions can be asserted based on what that optimization does. So, given two optimizations O_1 and O_2, we hope that causality based semantics can be used to address the question of the dependencies between O_1 and O_2. The results of this paper would be particularly relevant for optimizers that can be written as programs in the cc paradigm — such as the constraint based program analysis techniques, see for example [Hei92].

Modeling application-level causality Another reason for considering a richer semantic model for CCP arises from a class of applications for which CCP is being used.

One of the distinguishing characteristics of CCP is that it combines a powerful and expressive language for concurrent systems with a declarative reading. This makes it particularly attractive for use in modeling (concurrent) physical systems. The Model-based Computing project [FS95] is developing models for reprographics systems (photo-copiers) and their components. From these physics-based models, reasoners are used to derive information that can be plugged into standard architectures for tasks such as simulation and scheduling. Each physical component is modeled as a transducer which accepts inputs and control signals, operates in a given set of modes, and produces output signals. Models of assemblies are put together by connecting models of components in the same way as the components are put together to form the assemblies.

In such a context, the task of *scheduling* is to determine the control signals and the inputs which should be supplied to the system so as to *cause* the production of the given output. In other words, given a program P (the system model), and given constraints $o_1, \ldots, o_n$ (on the output variables), it is desired to produce constraints $i_1, \ldots, i_n$ on the input and control variables such that P can coherently (i.e. with the same set of choices for resolving indeterminacy) produce o_1 when run from i_1, o_2 when run from i_2 and so on. Generally one is interested in "minimal" explanations, i.e., the weakest i_j that can produce o_j.

One can recover explanations from the standard semantics as follows. Find t in the denotation of P such that t when run on i_j produces o_j for each j. Note that each t corresponds to a coherent execution of the program. However, in general the standard semantics will not able to provide minimal explanations.

Example 1.1 (Faulty not gates) Consider a not gate that may be arbitrarily be in one of three modes, `ok`, `stuck_at_1` or `stuck_at_0`:

```
not(Mode, In, Out) ::
   (Mode = ok→Out = not(In)
    □ Mode = stuck_at_1→Out = 1
    □ Mode = stuck_at_0→Out = 0).
```

Intuitively, one may say that for this gate, `Mode = ok` *causes* the output to be the negated version of the input, etc.

Consider, now an assembly P of two disconnected not gates `not(M1, X1, Y1)`, `not(M2, X2, Y2)`. An explanation that can be offered for $\mathtt{o}_1 = (\mathtt{Y1} = \mathtt{1})$ and $\mathtt{o}_2 = (\mathtt{Y2} = \mathtt{0})$ are $\mathtt{i}_1 = \mathtt{i}_2 = (\mathtt{M1} = \mathtt{ok}, \mathtt{X1} = \mathtt{0}, \mathtt{M2} = \mathtt{ok}, \mathtt{X2} = \mathtt{1})$.

However, it is quite clear that the corresponding minimal explanations are $\mathtt{i}_1 = (\mathtt{M1} = \mathtt{ok}, \mathtt{X1} = \mathtt{0})$ and $\mathtt{i}_2 = (\mathtt{M2} = \mathtt{ok}, \mathtt{X2} = \mathtt{1})$; there is no causal relationship between $Y2$ and $O2$, and hence between the two sub-explanations. The standard semantics will however find *one* execution of the system that can answer both queries, and hence produce an *interleaved* answer that does not respect the causality in the program. □

1.2 Representing causality: the basic idea

From these considerations we are motivated to find a semantics for CCP that invalidates the Law of Interleaving.

Let us reconsider the basic notion of observation. What is the finer-grain structure in the store that we might observe? Given the discussion of the previous section, a natural idea (e.g., see [MR95]) is to associate a constraint c in the store with its *causes*, that is, with the constraints d that needed to be supplied by the environment in order to trigger some computation in the program that results in c. Thus the store can be taken to be a collection of such *contexted* constraints c^d, given the (usually implicit) program P. Such an assertion is read as "d causes c", with d the *cause*, and c the *effect*. A *run* of the program can thus be taken to generate multiple such contexted constraints in the store. Given such a collection of contexted constraints ρ, their associated *generated effect* is obtained by simply taking the conjunction of the effects of each assertion in ρ; in this way one may recover the "constraint store" of the usual operational semantics of CCP.

Example 1.2 (Contd) Consider the program P of Example 1.1, started in the presence of the constraints $\mathtt{M1} = \mathtt{ok}, \mathtt{M2} = \mathtt{ok}, \mathtt{X1} = \mathtt{0}, \mathtt{X2} = \mathtt{1}$. The activation of the behaviour of the two `not` agents yields the addition of $\mathtt{Y1} = \mathtt{not(X1)}^{\mathtt{M1=ok}}$ and $\mathtt{Y2} = \mathtt{not(X2)}^{\mathtt{M2=ok}}$ to the store. In the presence of the constraint $\mathtt{X1} = \mathtt{0}$, it should be possible to use the first assertion to derive: $\mathtt{Y1} = \mathtt{1}^{\mathtt{M1=ok,X1=0}}$. Similarly for $\mathtt{Y2} = \mathtt{0}$. □

This example also illustrates another important point about the causal execution of agents. What we wish to record in the store are the assumptions *on the environment* that were made in the production of a given contraint c. Conventionally, only "closed" programs are executed — that is, no interaction with the environment is allowed. In such cases, the resulting store of contexted constraints will contain no more information than can be gleaned from examining the generated effect. The possibility of interesting non-trivial differences arises when we internalize the interactions $c_1, \ldots, c_n$ with the environment by running the program P in parallel with the contexted constraints $c_1^{c_1}, \ldots, c_n^{c_n}$. Intuitively, c^c captures the notion that c is an "external" input. For a collection of constraints $c_1, \ldots, c_n$, define $\Uparrow (c_1, \ldots, c_n)$ to be the contexted constraint $c_1^{c_1}, \ldots, c_n^{c_n}$. Thus the execution of the program P "started in the presence of the constraints $\mathtt{M1} = \mathtt{ok}, \mathtt{M2} = \mathtt{ok}, \mathtt{X1} = \mathtt{0}, \mathtt{X2} = \mathtt{1}$ is to be thought of as the execution of the agent $\mathtt{P}, \Uparrow (\mathtt{M1} = \mathtt{ok}, \mathtt{M2} = \mathtt{ok}, \mathtt{X1} = \mathtt{0}, \mathtt{X2} = \mathtt{1})$.

Another useful way to think of a contexted constraint b^a is as the assertion "on assumption a, program P produces output b". The constraints that a program is initially started in are "assumed", that is, are taken to depend only on themselves. One may now think of the operational semantics as manipulating constraints tagged with their assumptions, while maintaining the intuitive semantics of assumptions, e.g. as done by an Assumption-based Truth Maintenance System (ATMS) [deK86].

Logic of contexted constraints. Execution of a program is thus taken to yield a store of contexted constraints. However, the actual store that results depends in ways on the syntax of the program that are not crucial. For instance, the contexted store that results on the execution of $\mathtt{a} \parallel \mathtt{a} \rightarrow \mathtt{b}$ (in the presence of no other constraints) is different from that obtained from $\mathtt{a} \parallel \mathtt{b}$, though semantically they should be identical. Indeed, the need to abstract from the concrete syntax is already present in the standard semantics. Two programs A and B are considered behaviorally equivalent for a given initial store c if they produce stores d and e respectively that are equivalent (even if they are syntacticaly distinct), i.e. they *entail* each other. The entailment relation on constraints relevant there is the primitive relation $\vdash_{\mathcal{C}}$ supplied with the constraint system. Given that the store is now taken to contain contexted constraints, what is the relevant entailment relation?

Consider the judgement:

$$o_1, \ldots, o_n \vdash o$$

where o ranges over contexted constraints b^a. Such a judgement should be taken to hold exactly when it is the case that for any program P, any run of P which satisfies the assertions $o_1, \ldots, o_n$ could also satisfies o. In the following, let Γ, Δ range over multisets of o.

From elementary considerations it is clear that the following structural rules should hold:

$$\frac{\Gamma, p, q, \Delta \vdash o}{\Gamma, q, p, \Delta \vdash o} \quad \frac{\Gamma \vdash o}{\Gamma, p \vdash o} \quad \frac{\Gamma, p, p \vdash o}{\Gamma, p \vdash o}$$

Regarding the identity rules, it is clear that any program P should be able to produce d on the assumption c if in fact c entails d given the underlying constraint system. And we should expect that the Cut rule should hold:

$$\frac{a \vdash_{\mathcal{C}} b}{\vdash b^a} \quad \frac{\Gamma \vdash d \quad \Delta, d \vdash o}{\Gamma, \Delta \vdash o}$$

Finally we are left with the rules for inference that involve basic observations. Assume (1) that a process P satisfies all the assertions in Γ together with b^a, (2) $\Gamma \vdash a^{true}$, and that (3) for any process it is the case that if it satisfies Γ, and b^{true}, then it must satisfy o. From (1) and (2) it follows that P can on its own (i.e. assuming only that the environment supplies `true`), produce a. However, from (1), P satisfies b^a; therefore it must be the case that P can, on its own, produce b, and hence P satisfies b^{true}. But then, by (3) P satisfies o. This leads to the validity of the inference rule:

$$\frac{\Gamma \vdash a^{true} \quad \Gamma, b^{true} \vdash o}{\Gamma, b^a \vdash o}$$

Now it remains to consider the conditions under which it can be established that any program P satisfying Γ must satisfy b^a. Assume that for any program P it is the case that if P satisfies Γ and can on its own produce a, then it can on its own produce b. Now assume that Q is a program that satisfies Γ. Now note that if Q satisfies Γ, Q, a must also satisfy Γ. Clearly Q, a can on its own produce a. Therefore, by assumption Q, a must on its own be able to produce b. But if Q, a can produce b, then it must be the case that when Q is supplied a by the environment it can produce b. Hence we have:

$$\frac{\Gamma, a^{true} \vdash b^{true}}{\Gamma \vdash b^a}$$

Thus in our analysis, the "internal" logic of causation turns out to be that of intuitionistic implicational logic (over the underlying constraint system).

Denotational semantics. The observations of a program P are thus taken to be the contexted stores generated when P is executed in the presence of different constraints, modulo the equivalence generated by $\vdash$. What should a denotational semantics that respects this notion of observation look like?

To answer this, let us return to the analysis of the interaction between a system S and its environment E, via shared variables V. Instead of thinking of S as engaged in a *sequence* of interactions with E (e.g., detecting the presence of a constraint c_1 and producing c_2, *and then* detecting the presence of constraints c_3 and producing c_4 and so on), one should now also allow the possibility of *several* such independent interactions with the environment. That is, one should allow for interactions as described by the richer grammar:

$$k ::= c \mid c \to k \mid k \wedge k$$

Each k is called a *closure*; as before the conjunction of all constraints appearing in k, denoted $|k|$ is called its *bound*. Mathematically, each such k can be taken to describe a certain class of finitary closure operators over the lattice generated by the constraint systems, called *bounded closure operators* (bco's). Intuitively, closure operators allow for parallel branches of causality, whereas trace operators sequentialize these branches. Thus closure operators serve for CCP the role that "pomsets" serve for true concurrency semantics for other languages. (More precisely, multiplicities of constraints are irrelevant in CCP, since conjunction is idempotent. The poset of interest R can be recovered from the closure operator f by: $c\,R\,d$ iff $f(d)$ entails $f(c)$.) The denotation of an agent may then be taken to be the set of all bco's that an agent can engage in. We will show that program combinators can be defined over such a structure, and in fact such a denotational semantics is fully abstract with respect to observing the contexted constraints in the final store. As before, the denotation of a determinate program P is equivalent to the parallel composition of each of the bco's that can be observed of P; interestingly, however, this denotation is identical to that which would be obtained in the standard semantics. Thus, in some sense, the standard analysis of determinate CCP already incorporates an analysis of causality.

Rest of this paper The rest of this paper is concerned with fully developing these notions. We give the precise "causal" operational semantics and develop the denotational semantics along the lines sketched above. We study two kinds of models of indeterminacy, corresponding to must and may testing in the sense of [dNH84], and using the Hoare [Plo76] and the Smyth powerdomain[Smy78] respectively to handle indeterminacy. We establish full abstraction results. In addition, we expose some of the logical character of CCP by presenting two sound and complete proof systems that can be used to establish that an observation lies in the Hoare (resp. Smyth) denotations of a program P.

Related work

Many "true concurrency" semantics, see for example [AH89,BCHK92,vGV87,Gor91,Pra86, Vog92,Win87], which capture causality to varying degrees, have been proposed for other models of concurrency, including process algebras such as CSP and CCS, Petri Nets, and event structures. These semantics have typically generalized interleaving semantics to encode some degree of concurrency, such as "steps" of concurrent actions rather than single actions, and "pomsets" of partially-ordered multisets of actions rather than linear temporal sequences. Note that while our semantics is a sets of closure operators semantics, it does make the early vs. late branching distinction, so $a \rightarrow ((b \rightarrow B) \,\square\, (c \rightarrow C)) \neq (a \rightarrow b \rightarrow B) \,\square\, (a \rightarrow c \rightarrow C)$, unlike [Pra86].

The studies most relevant to the present paper are studies of causality and true concurreny issues in the cc paradigm — see for example, [MR95,MR91,dBGMP94,dBPB95].

[MR95,MR91] propose a framework, based on graph rewriting and occurrence nets, to study true concurrency issues in the cc languages [MR95]. In this paper, we do (essentially) adopt the framework of "contextual agents" of [MR95,MR91] to describe extraction of causality information from the program execution in the operational semantics. Our primary distinct contribution is the logical/denotational analysis of the operational semantics.

[dBPB95] propose a true concurrency framework for a more general class of non-monotonic cc languages. When specialised to monotonic cc languages, their framework yields essentially a "step semantics", where a collection of concurrent actions can be performed at each step. Our work differs from [dBPB95] in the analysis of the process of addition of constraints. [dBPB95] distinguishes different occurrences of the same constraint. This view of separating occurrences of the same constraint is not appropriate for some of our motivating examples, especially the scheduler. In our framework, the process of imposition of constraints is idempotent.

[dBGMP94] adapts the study of the logical structure of domains [Abr91] to CCP. That paper does not directly address issues of causality and true concurrency; however, we acknowledge the methodological influence of their work on our work.

2 Causal Transition System for CCP

cc languages are described parametrically over a *constraint system*. The constraint system determines the pieces of partial information that can be added to the store. For a detailed description, we refer the reader to [SRP91,Sar92]. Briefly, a constraint system $\mathcal{C}$ consists of a set D of first order formulas called primitive constraints or *tokens* and an entailment relation $\vdash_{\mathcal{C}} \subseteq Fin(D) \times D$ between them. This relation tells us which tokens follow from which others, and should be decidable and finitary, i.e. the set $Fin(D)$ consists of finite sets of tokens.

Subsets of D closed under $\vdash_{\mathcal{C}}$ are called *constraints*. The set of all constraints, denoted $|D|$, is ordered by inclusion, and forms an algebraic lattice. We will use $\sqcup$ and $\sqcap$ to denote joins and meets of this lattice. Note that this ordering corresponds to the information ordering on tokens, as stronger constraints entail more tokens. We will use $a, b, \ldots$ to denote tokens, and $c, d, e, \ldots$ to denote constraints.

The syntax of CCP languages is as follows:

$$\boxed{\begin{array}{lcl} P & ::= & a \mid P \parallel P \mid \square_i a_i \rightarrow P_i \mid \exists X.P \mid g(X) \mid \{D.P\} \\ D & ::= & \epsilon \mid g(X) :: P, D \end{array}}$$

We use the words "agents" and "programs" synonymously in the rest of the paper. Formally, the execution semantics can be given by a transition system. A *configuration* Γ is a multiset of agents. $\sigma(\Gamma)$ denotes the set of tokens in Γ. The transition system relates configurations to configurations and is the least relation satisfying the following axioms and inference rules [SRP91]:

$$\Gamma, (P \parallel Q) \longrightarrow \Gamma, P, Q \qquad \frac{\sigma(\Gamma) \vdash_{\mathcal{C}} a_i}{\Gamma, (\,\square_{i \in I} a_i \rightarrow P_i) \longrightarrow \Gamma, P_i}$$

$$\Gamma, \exists X.P \longrightarrow \Gamma, P[Y/X] \quad (Y \text{ new}) \quad \Gamma, g(X) \longrightarrow \Gamma, P(X) \quad \text{if } g(X) :: P \in D$$

Observing causality. To detect causality information, we will allow our configurations to remember the set of constraints which enabled each agent to be executed, i.e., instead of taking a configuration to be a multiset of agents, we take it to be a multiset of contexted agents P^a. The transition relation may now be defined straightforwardly:

$$\Gamma, (P \parallel Q)^c \longrightarrow \Gamma, P^c, Q^c \qquad \frac{\sigma(\Gamma) \vdash_{\mathcal{C}} a_i}{\Gamma, (\,\square_{i \in I} a_i \rightarrow P_i)^c \longrightarrow \Gamma, P_i^{a_i, c}}$$

$$\Gamma, (\exists X.P)^c \longrightarrow \Gamma, P[Y/X]^c \quad (Y \text{ new}) \quad \Gamma, g(X)^c \longrightarrow \Gamma, P(X)^c \quad \text{if } g(X) :: P \in D$$

Operational semantics From the transition system, various operational semantics can be defined in the obvious way. Each operational semantics records for a program the contexted store that results when a program is executed in the presence of different assumed constraints. The different operational semantics arise from different treatment of non-termination (should only terminated computations be observed?) and indeterminacy (when are two sets of results equivalent?). For the purposes of this paper we shall focus on the "may" semantics which observes for each program P the results of terminating execution sequences, for all possible assumed constraints. In Section 4 we briefly discuss a similar treatment for "must" semantics, which leads to the Smyth powerdomain.

Note that in order to accomodate hiding, our observations will need to hide all the new variables introduced in the transitions. Thus they will be of the form $\delta V.o$, where $\delta V.o = \exists X_1. \ldots \exists X_n.o$, for all X_i in $\mathbf{var}(o) - V$. As none of these X_i's will occur in $\mathbf{var}(P, c)$, they need not be in the cause of any contexted constraint in o. Now we can use the law $\exists_X(a^b, c^d) = (\exists_X a)^b, (\exists_X c)^d, (\exists_X a \wedge c)^{\{b,d\}}$ to get the usual contexted constraints. The logical rules for existential quantification of observations are the usual intuitionistic rules for existential quantification. Let $\rho(\Gamma)$ be the multiset of contexted constraints in Γ.

The may semantics may be defined straightforwardly:

$$\mathcal{O}_H[\![P]\!] = \{\delta V.\rho(\Gamma) \mid \exists c.\ P, \Uparrow(c) \rightarrow^* \Gamma \not\rightarrow, V = \mathbf{var}(P, c)\}$$

3 Causal denotational semantics

A *closure operator* is a function f from constraints to constraints, which is extensive ($f(c) \supseteq c$), idempotent ($f \circ f = f$) and monotone (if $c \supseteq d$ then $f(c) \supseteq f(d)$). An alternative way of presenting a closure operator is as a set of its fixed points, *i.e.* those constraints c such that $f(c) = c$. We recall that the set of fixed points of a closure operator is a set of constraints which is closed under greatest lower bounds (glb's) — any set of constraints A which is closed under glb's can be used to define a closure operator by $f_A(c) = \sqcap\{d \in A \mid d \supseteq c\}$. In the rest of this paper we will use both these representations interchangeably.

A *bounded closure operator* (bco) is a pair (f, d), where f is a closure operator, and d is a constraint, $d \in f$. The d determines the *domain* of the bco — this is defined as $d\downarrow$, the set of constraints smaller than d. Thus if $(f, d), (g, d)$ are bco's, with $f \cap d\downarrow = g \cap d\downarrow$, then we consider $(f, d) = (g, d)$.

Closure operators are ordered pointwise — thus if for all c, $f(c) \subseteq g(c)$, we define $f \leq g$. In the set representation, this simply becomes $f \leq g$ iff $f \supseteq g$. We will refer to this ordering as the *information ordering*, it is the converse of the usual set ordering. This ordering is extended to bco's — $(f, d) \leq (g, d)$ iff $f \leq g$. Note that we do not compare bco's with different domains, these are regarded as unrelated. The set of all bco's under this ordering now forms a domain, called **Obs**, the carrier set of this domain is denoted as $\mid$ **Obs** $\mid$.

In the rest of this section we define the semantics of cc languages by examining the Hoare powerdomain constructions on this domain[Plo76]. Later in section 4 we will examine the Smyth powerdomain construction. In this paper, we use the representation theorems of [Smy78] to simplify the presentation of the powerdomain constructions. Furthermore, for powerdomain aficionados, we point out that we are technically using the powerdomain constructions with the emptyset, rather than the more traditional ones without the emptyset.

The elements of the Hoare powerdomain on **Obs** are sets S of bco's satisfying the closure condition:

$$(f, d) \in S, g \supseteq f \Rightarrow (g, d) \in S$$

The ordering relation is given by subset inclusion:

$$S_1 \sqsubseteq S_2 \Leftrightarrow S_1 \subseteq S_2$$

The Hoare powerdomain on **Obs** yields a complete lattice — the least element is the empty set, the greatest element is $\mid$ **Obs** $\mid$, least upper bounds are given by union, and greatest lower bounds are given by intersection.

The semantics of the various programs can be given as follows:

$$\begin{array}{rcl}
\mathcal{H}[\![a]\!] & \stackrel{d}{=} & \{(f, d) \in \mathbf{Obs} \mid d \supseteq a, f \supseteq a\uparrow\} \\
\mathcal{H}[\![P \parallel Q]\!] & \stackrel{d}{=} & \{(f \cap g, d) \in \mathbf{Obs} \mid (f, d) \in \mathcal{H}[\![P]\!], (g, d) \in \mathcal{H}[\![Q]\!]\} \\
\mathcal{H}[\![\,\Box_{i \in I} a_i \rightarrow P_i]\!] & \stackrel{d}{=} & \{(d\downarrow, d) \in \mathbf{Obs} \mid \forall i \in I. d \not\supseteq a_i\} \\
& & \cup \bigcup_{i \in I}\{(f, d) \in \mathbf{Obs} \mid d \supseteq a_i, \exists (f_i, d) \in \mathcal{H}[\![P_i]\!], f \supseteq a_i \rightarrow f_i\} \\
\mathcal{H}[\![\exists X.P]\!] & \stackrel{d}{=} & \{(f, d) \in \mathbf{Obs} \mid \exists (g, c) \in \mathcal{H}[\![P]\!]. \exists X.g = \exists X.f, \exists X.c = \exists X.d, \\
& & \qquad \forall e \subseteq c. g(\exists_X e) = g(e)\}
\end{array}$$

Recursive definitions are as usual defined by least fixed points.

Note the extra clause in the definition of $\exists X.P$, which states that $g(\exists X.c) = g(c)$ for all c in the domain of g. This is motivated by the fact that P cannot receive any information about X from the environment, thus any information it uses on X needs to be produced by it. So any observed bco in P which can be a witness for an observation in $\exists X.P$ must use only internally generated X-information, and so it must be the case that on any input X, it must be able to ignore the X information, by generating the same output on $\exists X.c$ as on c.

We note a few interesting facts to give the reader some more intuition about the resulting semantics:

- Let P_i be a collection of processes indexed by i. Let a be a token in the constraint system. Then $\mathcal{H}[\![a \rightarrow P_i]\!] \sqsubseteq \mathcal{H}[\![\, \Box_{i\in I} a \rightarrow P_i]\!]$. We justify the inequation by noting that the semantic interpretation of bounded choice in the special case when all the guards are identical is just union. This inequation is characteristic of Hoare powerdomain style semantics — adding more branches to a process moves it up the ordering in the powerdomain.

- Let the process P_1 be defined recursively as $P_1 :: P_1$. Let the process P_2 be defined recursively as $P_2 :: b \parallel P_2$. Then $\mathcal{H}[\![P_1]\!] = \mathcal{H}[\![P_2]\!] = \emptyset$. This indicates the treatment of termination by the semantics — only terminating runs are counted.

- Let P be any process. Consider the processes $P_3 = \Box\, [a \rightarrow P, a \rightarrow P_1]$, and $P_4 = \Box\, [a \rightarrow P, a \rightarrow P_2]$, where P_1, P_2 are as above. Then $\mathcal{H}[\![P_3]\!] = \mathcal{H}[\![P_4]\!] = \mathcal{H}[\![a \rightarrow P]\!]$. This further clarifies termination issues in the Hoare semantics — non-terminating runs are ignored.

Full abstraction We now show that the semantics is fully abstract with respect to the may operational semantics. We first state a lemma showing that if we embed the operational semantics of a program in the Hoare powerdomain in the obvious way, we get the denotational semantics of the program. The proof is by induction on the structure of the program. We use $[\![o]\!]$ to denote the closure operator associated with the observation o. Also, define $\|o\|$, the size of o, as the lub of all the constraints occurring in it — $\|c^d\| = c \sqcup d$, $\|o_1 \wedge o_2\| = \|o_1\| \sqcup \|o_2\|$.

Lemma 3.1 *For any program P, if $o \in \mathcal{O}_H[\![P]\!]$, then $([\![o]\!], \|o\|) \in \mathcal{H}[\![P]\!]$. Conversely, if $(f, d) \in \mathcal{H}[\![P]\!]$, then there is an $o \in \mathcal{O}_H[\![P]\!]$ such that $\|o\| = d$, and $[\![o]\!] \subseteq f$.*

From this we get the full abstraction theorem for the Hoare semantics —

Theorem 3.2 *If P, Q are two indeterminate programs, and $\mathcal{H}[\![P]\!] \neq \mathcal{H}[\![Q]\!]$, then there is an input a, such that the possible output contexted stores of $P \parallel a$ and $Q \parallel a$ are different. Conversely, if the set of possible output contexted stores of P and Q are different for some input, then $\mathcal{H}[\![P]\!] \neq \mathcal{H}[\![Q]\!]$.*

3.1 Logical form

We now present the Hoare semantics in a logical form, based on Abramsky's [Abr91]. We observe the properties that are true of the program, by executing the program. These properties are used to construct the denotational semantics of the program. This gives us an alternative presentation of the denotational semantics, and gives a clear connection between the operational and denotational semantics.

Properties are generated by the following syntax:

$$\phi ::= c \mid c \rightarrow \phi \mid \phi \wedge \phi \mid \exists_X \phi$$

We define the size of a property as the lub of all the constraints occurring in it — $\|c\| = c, \|c \to \phi\| = c \sqcup \|\phi\|, \|\phi \wedge \psi\| = \|\phi\| \sqcup \|\psi\|, \|\exists X.\phi| = \exists_X \|\phi\|$. Note that size is a syntactic property, and can be changed by appending $d \to d$ to any property (without changing its logical content).

Intuitively, the Hoare semantics of a program consists of all the properties which are satisfied exactly by some execution sequence of the program in some input context. Define an execution sequence of a program P with assumptions c as a sequence $S = (P_0, P_1, P_2, \ldots, P_n)$, where $P_0 = (P, \Uparrow(c))$, $P_n \not\longrightarrow$ and for each i between 0 and n, $P_i \longrightarrow P_{i+1}$. An execution sequence $S = (P_0, P_1, P_2, \ldots, P_n)$ satisfies ϕ iff $\delta V(\rho(P_n)) \vdash_{IL} \phi$, and $\|\phi\| = \|\delta V(\rho(P_n))\|$. Here $\vdash_{IL}$ denotes Intuitionistic logic implication, which is the logic of determinate cc programs.

In the following, Γ, Γ' represent multisets of agents.

$$\frac{\Gamma, A, B, \Delta \Vdash \phi}{\Gamma, B, A, \Delta \Vdash \phi} \qquad \frac{\Gamma \Vdash \phi \quad \phi \vdash_{IL} \psi \quad \|\phi\| = \|\psi\|}{\Gamma \Vdash \psi}$$

$$\frac{c_1, \ldots, c_n \vdash_{IL} \phi \quad \|\phi\| = \|c_1\| \sqcup \ldots \sqcup \|c_n\|}{c_1, \ldots, c_n \Vdash \phi}$$

$$\frac{\Gamma, A, B \Vdash \phi}{\Gamma, (A \parallel B) \Vdash \phi} \qquad \frac{\Gamma \Vdash \phi \quad \Gamma' \Vdash \psi \quad \|\phi\| = \|\psi\|}{\Gamma, \Gamma' \Vdash \phi \wedge \psi}$$

$$\frac{\sigma(\Gamma) \vdash c_i \quad c_i, \Gamma, A_i \Vdash \phi}{\Gamma, \Box_{i \in I} c_i \to A_i \Vdash \phi} \qquad \frac{\Gamma, c \Vdash \phi}{\Gamma \Vdash c \to \phi}$$

$$\frac{\forall i \in I. d \not\supseteq c_i \quad \Gamma \Vdash d}{\Gamma, \Box_{i \in I} c_i \to A_i \Vdash d} \qquad \frac{\Gamma, A(X) \Vdash \phi \quad g(X) :: A(X)}{\Gamma, g(X) \Vdash \phi}$$

$$\frac{\Gamma, P[Y/X] \Vdash \phi \quad (Y \text{ new in } \Gamma, \phi)}{\Gamma, \exists X.P \Vdash \phi} \qquad \frac{\Gamma, \Vdash \phi[t/X]}{\Gamma \Vdash \exists X.\phi}$$

We can now show the theorem that shows the correspondence between the denotation of a process and its logic, its proof is by induction over the structure of programs.

Theorem 3.3 *$P \Vdash \phi$ iff $(\llbracket\phi\rrbracket, \|\phi\|) \in \mathcal{H}\llbracket P\rrbracket$, where $\llbracket\phi\rrbracket$ is the closure operator corresponding to ϕ.*

4 Causal Must semantics

An alternative causal semantics of cc programs allows us to observe what *must* be true for all runs of the system — this style does not ignore infinite runs of the system, and allows us to observe intermediate results. The formulation is quite similar to the may semantics presented above, so here we will review it briefly.

o is a *must* observation of a program P with context c if either $P, \Uparrow(c) \to^* \Gamma \not\to$ and $o = \delta V.\rho(\Gamma)$, where $V = \mathbf{var}(P, c)$ or there is an infinite sequence $P, \Uparrow(c) \to \Gamma_1 \to \Gamma_2 \to \ldots$, such that $o = \bigcup_i \{\delta V.\rho(\Gamma_i)\}$. We say that o is minimal for P, c if there is no other must observation o' such that $o \vdash o'$. This condition will allow us to identify two sets of observations which have the same minimal elements, giving us the must tests of [dNH84].

$$\mathcal{O}_S\llbracket P\rrbracket = \{o \mid \exists c.\ o \text{ is a must observation of } P \text{ in context } c, \text{ and } o \text{ is minimal for } P, c\}$$

The Smyth powerdomain. The elements of the Smyth powerdomain on **Obs** are sets S of bco's satisfying the condition:

$$(f,d) \in S, g \subseteq f \Rightarrow (g,d) \in S$$

In the Smyth powerdomain, the ordering relation is given by reverse subset inclusion:

$$S_1 \sqsubseteq S_2 \Leftrightarrow S_1 \supseteq S_2$$

The Smyth powerdomain on **Obs** yields a complete lattice — the greatest element is the empty set, the least element is $\mid \mathbf{Obs} \mid$, least upper bounds are given by intersection, and greatest lower bounds are given by union.

The semantics of the various programs can be given as follows:

$$\begin{array}{rcl}
\mathcal{S}[\![a]\!] & \stackrel{d}{=} & \{(f,d) \in \mathbf{Obs} \mid f \subseteq a \uparrow\} \\
\mathcal{S}[\![P \parallel Q]\!] & \stackrel{d}{=} & \mathcal{S}[\![P]\!] \cap \mathcal{S}[\![Q]\!] \\
\mathcal{S}[\![\,\Box_{i \in I} a_i \rightarrow P_i]\!] & \stackrel{d}{=} & \{(f,d) \in \mathbf{Obs} \mid \forall i \in I. d \not\supseteq a_i\} \\
 & & \cup \bigcup_{i \in I} \{(f,d) \in \mathbf{Obs} \mid d \supseteq a_i, \exists (f_i, d) \in \mathcal{S}[\![P_i]\!], f \subseteq a_i \rightarrow f_i\} \\
\mathcal{S}[\![\exists X.P]\!] & \stackrel{d}{=} & \{(f,d) \in \mathbf{Obs} \mid \exists (g,c) \in \mathcal{S}[\![P]\!]. \exists X.g = \exists X.f, \exists X.c = \exists X.d, \\
 & & \qquad \forall e \subseteq c. g(\exists_X e) = g(e)\}
\end{array}$$

Recursion occurs via least fixed points.

We note a few interesting facts to give the reader some more intuition about the resulting semantics: we are choosing the same examples as those discussed for the Hoare semantics, and draw the readers attention to the differences.

- Let P_i be a collection of processes indexed by i. Let a be a token in the constraint system. Then $\mathcal{S}[\![a \rightarrow P_i]\!] \sqsupseteq \mathcal{S}[\![\,\Box_{i \in I} a \rightarrow P_i]\!]$. This can be justified by noting that the semantic interpretation of bounded choice in the special case when all the guards are identical is just union. This inequation (which we note is the exact converse of the one for the earlier semantics) is characteristic of Smyth powerdomain style semantics — adding more branches to a process moves it lower down the ordering in the domain.

- Let the process P_1 be defined recursively as $P_1 :: P_1$. Then $\mathcal{S}[\![P_1]\!] = \mid \mathbf{Obs} \mid$. Let the process P_2 be defined recursively as $P_2 :: a \parallel P_2$. Then $\mathcal{S}[\![P_1]\!] = \mathcal{S}[\![a]\!]$. These examples indicate the treatment of non-termination by the semantics — in effect, the semantics only looks at the store as it evolves, and allows one to observe "intermediate" stores even in an unbounded computation.

- Let P be any process. Consider the process $P_3 = b \rightarrow P \,\Box\, b \rightarrow P_1$, where P_1 is as above. Then $\mathcal{S}[\![P_3]\!] = \mathcal{S}[\![b \rightarrow P_1]\!] = \mid \mathbf{Obs} \mid$. This further clarifies the treatment of non–determinism in the Smyth semantics. The intuitive reasoning is as follows: if b is not entailed by the store, neither side does anything to the store. If b is entailed by the store, the minimum guaranteed output is from the P_1 branch, which adds nothing new. This type of minimum guarantee reasoning is typical of Smyth powerdomain style semantics.

Similar to Lemma 3.2, we get a full abstraction theorem for the Smyth semantics —

Theorem 4.1 *If P, Q are two indeterminate programs, and $\mathcal{S}[\![P]\!] \neq \mathcal{S}[\![Q]\!]$, then there is a context a, such that the output stores of $P \parallel a$ and $Q \parallel a$ are different. Conversely, if the set of* minimal *output stores of P and Q are different, then $\mathcal{S}[\![P]\!] \neq \mathcal{S}[\![Q]\!]$.*

Smyth semantics in logical form. Intuitively, the Smyth semantics consists of properties that are satisfied by all runs of the program in some context. We will present a brief sketch here, and will include the complete development in the full paper. We redefine $P, c \Vdash \phi$ to mean that *every* execution sequence of P in the context c satifies ϕ. The set of properties is derived from the following grammar:

$$\phi ::= c \mid c \rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists X.\phi$$

The logical rules are omitted — they correspond to the transition rules on the left (taking *all* valid choices for the indeterminate choice operator), and standard intuitionistic rules on the right.

The logical semantics of a program can now be defined. Embed ϕ in the Smyth powerdomain as $\mathcal{E}_S(\phi)$ using the formulas given for the denotational semantics, treating $\vee$ as set union. Then the logical semantics is given as the intersections of the embedded sets — $\mathcal{L}_S[\![P]\!] = \bigcap\{\mathcal{E}_S(\phi) \mid P, c \Vdash \phi\}$. And we get the theorem:

Theorem 4.2 *For any indeterminate* cc *program P, $\mathcal{L}_S[\![P]\!] = \mathcal{S}[\![P]\!]$.*

5 Acknowledgements

We would like to thank Prakash Panangaden, Martin Rinard, Markus Fromherz and Saumya Debray for extensive discussions on the topics discussed in this paper, which led to many insights. Work on this paper was funded by grants from ONR and ARPA, and the second author was also funded by NSF.

References

[Abr91] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[AH89] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 138–145. IEEE Computer Society Press, 1989.

[BCHK92] G. Boudol, I. Castellani, Matthew C. Hennessy, and A. Kiehn. A theory of processes with localities. In *Proceedings of International Conference on Concurrency Theory, Volume 630 of Lecture Notes in Computer Science*, pages 108–122, 1992.

[dBGMP94] F. S. de Boer, M. Gabrielli, Elena Marchiori, and Catuscia Palamidessi. Proving concurrent programs correct. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–108, 1994.

[dBP91] F. S. de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings of TAPSOFT/CAAP*, pages 296–319, LNCS 493, 1991.

[dBPB95] F. S. de Boer, Catuscia Palamidessi, and Eike Best. Concurrent constraint programming with information removal. In *Proceedings of the Concurrent Constraint Programming Workshop*, pages 1–13, Venice, 1995.

[deK86] J. deKleer. An assumption based TMS. *Artifical Intelligence*, 28:127–162, 1986.

[dNH84] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[FS95] Markus Fromherz and Vijay Saraswat. Model-based computing: Using concurrent constraint programming for modeling and model compilation. In *Principles and Practices of Constraint Programming*, volume 976 of *LNCS*, pages 629–635. Springer Verlag, 1995.

[Gor91] Roberto Gorrieri. *Refinement, Atomicity, and Transactions for Process Description Languages.* PhD thesis, University of Pisa, 1991.

[Hei92] N. Heintze. *Set–Based Program analysis.* PhD thesis, Carnegie Mellon University, 1992.

[JPP91] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully-abstract semantics for a functional language with logic variables. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[MR91] Ugo Montanari and Francesca Rossi. True concurrency semantics for concurrent constraint programminmg. In V. Saraswat and K. Ueda, editors, *Proc. of the 1991 International Logic Programming Symposium*, 1991.

[MR95] Ugo Montanari and Francesca Rossi. A concurrent semantics for concurrent constraint programs via contextual nets. In *Principles and Pranctices of Constraint Programming*, pages 3–27, 1995.

[Plo76] G.D. Plotkin. A powerdomain construction. *SIAM J. of Computing*, 5(3):452–487, September 1976.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[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, 1992.

[Sar92] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

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

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

[vGV87] Rob van Glabbeek and Frits Vaandrager. Petri net models for algebraic theories of concurrency. In *Proceedings of PARLE, Volume 259 of the Lecture Notes in Computer Science*, pages 224–242, 1987.

[Vog92] Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *LNCS*. Springer-Verlag, 1992. 252 pp.

[Win87] Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Volume 255 of Lecture Notes in Computer Science*, pages 325–392, 1987.