ELSEVIER

Theoretical Computer Science 278 (2002) 223–255

Theoretical Computer Science

www.elsevier.com/locate/tcs

Truly concurrent constraint programming

V. Gupta[a,*], R. Jagadeesan[b], V.A. Saraswat[c]

[a] *Stratify Inc., 501 Ellis Street, Mountain View, CA 94043, USA*

[b] *Department of Computer Science, Loyola University Chicago, 6525 N. Sheridan Road, Chicago, IL 60626, USA*

[c] *41 Networks, 208 Morris Avenue, Mountain Lakes, NJ 07046, USA*

Abstract

We study "causality" relationships in Concurrent Constraint Programming: what is observed is not just the conjunction of constraints deposited in the store, but also the causal dependencies between these constraints. We describe a denotational semantics for cc that is fully abstract with respect to observing this "causality" relation on constraints. This semantics preserves more fine-grained structure of computation; in particular the Interleaving Law

$$(a \to P) \parallel (b \to Q) = (a \to (P \parallel (b \to Q)))(b \to (Q \parallel (a \to P)))$$

is not verified (is indeterminate choice). Relationships between such a denotational approach to true concurrency and different powerdomain constructions are explored. © 2002 Published by Elsevier Science B.V.

1. Introduction

Concurrent constraint programming [20, 23, 24] is a simple and powerful model of concurrent computation obtained by internalizing the notion of computation as deduction 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 primitive constraint or *token* (over a given finite set of variables) is a finitary specification of possibly partial information about the values the variables can take. A typical example of a token is a first-order formula over some algebraic structure. Tokens come naturally equipped with an entailment relation: $a_1, \ldots, a_n \vdash a$ holds exactly if the presence of tokens $a_1, \ldots, a_n$ implies the presence of the token a. Thus tokens can combine additively, without any prejudice about their source, to produce other tokens. An *agent* has access to a finite

* Corresponding author.

E-mail addresses: vineet@stratify.com (V. Gupta), radha@cs.luc.edu (R. Jagadeesan), vijay@saraswat.org (V.A. Saraswat).

0304-3975/02/$ - see front matter © 2002 Published by Elsevier Science B.V.

PII: S0304-3975(00)00337-6

set of variables – the basic operations it may perform are to constrain some subset of variables it has access to by posting a token ($A ::= a$), to check whether a token is entailed by ones that have already been posted and if so, reduce to another agent, perhaps non-deterministically ($A ::= \Box_{i\in I} a_i \to A_i$), to create 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 cc languages [5, 24]. It is natural to observe for every agent the 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 tokens a_1 (on V), producing tokens a_2, and then suspending until more tokens a_3 are produced, and then producing tokens a_4, and so on. Finite sequences of such interactions may be described by means of the grammar

$$t ::= a \mid a \to t \mid a \wedge t$$

Each t is called a *trace*; the conjunction of all tokens 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 bto's that an agent can engage in. Saraswat et al. [24] 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 cc can in fact be described by a single closure operator, equivalent to the parallel composition of each of the bto's. In what follows, we will call this the "standard model" of cc.

It is important to point out that the nature of communication in cc 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 cc.[1] For instance, if a token c is logically equivalent to the conjunction of tokens a and b, the standard model validates the *Law of Non-Atomicity of Tells*

$$a \parallel b = c \tag{1}$$

and the *Law of Non-Atomicity of Asks*

$$a \to b \to A = c \to A \tag{2}$$

In addition the standard model also validates the Interleaving Law:

$$(a \to P) \parallel (b \to Q) = (a \to (P \parallel (b \to Q))) \Box (b \to (Q \parallel (a \to P))) \tag{3}$$

[1] For the purposes of this paper, cc refers to "eventual tell" version of cc [20], rather than the "atomic tell" version for which a similar claim cannot be made.

1.1. Why causality?

Such a development of the semantics of cc is not fully satisfactory for modeling application-level causality.

One of the distinguishing characteristics of cc 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 [9] is developing models for reprographics systems (photocopiers) 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 be 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.

Now consider an assembly P of two disconnected not gates `not(M1,X1,Y1)` and `not(M2,X2,Y2)`. An explanation that can be offered for $o_1 = (Y1 = 1)$ and $o_2 = (Y2 = 0)$ is $i_1 = i_2 = (M1 = ok,\ X1 = 0,\ M2 = ok,\ X2 = 1)$.

However, it is quite clear that the corresponding minimal explanations are $i_1 = (M1 = ok,\ X1 = 0)$ and $i_2 = (M2 = ok,\ X2 = 1)$; 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. An overview of our approach

1.2.1. Representing causality: the basic idea

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 [15]) is to associate a token a in the store with its *causes*, that is, with the token b that needed to be supplied by the environment in order to trigger some computation in the program that results in a. Thus the store can be taken to be a collection of such *contexted* tokens a^b, given the (usually implicit) program P. Such an assertion is read as "b causes a", with b the *cause*, and a the *effect*. A *run* of the program generates many such contexted tokens in the store. Given such a collection of contexted tokens 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 cc.

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

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 token a. Conventionally, only "closed" programs are executed – that is, no interaction with the environment is allowed. In such cases, the resulting store of contexted tokens 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 $a_1,\ldots,a_n$ with the environment by running the program P in parallel with the contexted tokens $a_1^{a_1},\ldots,a_n^{a_n}$. Intuitively, a^a captures the notion that a is an "external" input. For a collection of tokens $a_1,\ldots,a_n$, define $\Uparrow(a_1,\ldots,a_n)$ to be the set of contexted tokens $a_1^{a_1},\ldots,a_n^{a_n}$. Thus the execution of the program P "started in the presence of the tokens $\mathtt{M1=ok}$, $\mathtt{M2=ok}$, $\mathtt{X1=0}$, $\mathtt{X2=1}$" is to be thought of as the execution of the agent $\mathtt{P}$, $\Uparrow(\mathtt{M1=ok},\ \mathtt{M2=ok},\ \mathtt{X1=0},\ \mathtt{X2=1})$.

Another useful way to think of a contexted token b^a is as the assertion "on assumption a, program P produces output b". The tokens in the store 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 tokens tagged with their assumptions, while maintaining the intuitive semantics of assumptions, e.g. as done by an assumption-based truth maintenance system (ATMS) [7].

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:

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

or the Law of Intermediate Causation:

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

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

1.2.2. Logic of contexted tokens

Execution of a program is thus taken to yield a store of contexted tokens. 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 $\mathtt{a} \parallel \mathtt{a} \to \mathtt{b}$ (in the presence of no other tokens) is different from that obtained from $\mathtt{a} \parallel \mathtt{b}$, though semantically they should be identical, since no additional assumptions were needed of the environment to produce b. 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 a if they produce stores a_1 and a_2, respectively, that are equivalent (even if they are syntactically distinct), i.e. they *entail* each other. The entailment relation on tokens relevant there is the primitive relation $\vdash_{\mathscr{D}}$ supplied with the constraint system. Given that the store is now taken to contain contexted tokens, what is the relevant entailment relation?

The answer follows from what it means for a program to produce a contexted token. To produce a^b, the program produces a if it receives b. Thus these tokens behave like cc programs, so the "internal" logic of causation turns out to be that of intuitionistic implicational logic (over the underlying constraint system).

1.2.3. 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 tokens, 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 token a_1 and producing a_2, *and then* detecting the presence of a_3 and producing a_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:

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

Each k is called a *closure*; as before the conjunction of all tokens 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 cc the role that "pomsets" serve for true concurrency semantics for other languages. (More precisely, multiplicities of tokens are irrelevant in cc, since conjunction is idempotent. The poset of interest $\leqslant_R$ can be recovered from the closure operator f by: $a \leqslant_R b$ iff $f(b)$ entails $f(a)$.) The denotation of an agent may then be taken to be the set of all bco's that an agent can engage in. (In order to define recursion, we will define bco's over constraints rather than tokens; see Section 4.) 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 tokens in the final store. As before, the denotation of a determinate program P is equivalent to the parallel composition of the contexted tokens 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 cc already incorporates an analysis of causality; we extend this analysis to indeterminate programs.

1.3. Rest of this paper

The rest of this paper is concerned with fully developing these notions. We first review some background material on constraint systems and cc operational semantics. We then give the precise "causal" operational semantics, develop its properties and formally define equivalence of contexted tokens. We then develop the denotational semantics along the lines sketcted above. We study in detail a model of indeterminacy, corresponding to may testing in the sense of [8], and using a powerdomain construction motivated by the relational [16] powerdomain. We describe some equational laws to expose the flavor of the semantics and establish full abstraction results. In addition, we expose some of the logical character of cc by presenting a sound and complete proof system that can be used to establish that an observation lies in the denotations of a program P. In the following section, we also establish these results for a different model of indeterminacy, corresponding to must testing in the sense of [8], and using the Smyth powerdomain [22] to handle indeterminacy.

1.4. Related work

Many "true concurrency" semantics, such as [2, 3, 25, 10, 18, 26, 27], capture causality to varying degrees in the context of 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 $a \to ((b \to B) \square (c \to C)) \neq (a \to b \to B) \square (a \to c \to C)$, unlike [18].

The studies most relevant to the present paper are studies of causality and true concurrency in the cc paradigm – e.g. [4, 14, 15, 5]. Montanari and Rossi [14, 15] propose a framework, based on graph rewriting and occurrence nets, to study true concurrency issues in the cc languages [15]. In this paper, we do (essentially) adopt the framework of "contextual agents" of [14, 15] 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.

de Boer and Palamidessi [5] propose a true concurrency framework for a more general class of non-monotonic cc languages. When specialized 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 [5] in the analysis of the process of addition of constraints. de Boer and Palamidessi [5] 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.

de Boer et al. [4] adapts the study of the logical structure of domains [1] to cc. 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. Constraint systems

cc languages are described parametrically over a *constraint system* [19, 24]. For technical convenience, the presentation here is a slight variation on earlier published presentations.

The information added to the store consists of primitive constraints which are drawn from a *constraint system*. A constraint system $\mathscr{D}$ is a system of partial information, consisting of a set of primitive constraints (first-order formulas) or *tokens* D, closed under conjunction and existential quantification, and an inference relation (logical entailment) $\vdash$ that relates tokens to tokens. We use $a, b, c \ldots$ to denote tokens. $\vdash$ naturally induces logical equivalence, written $\approx$. Formally,

Definition 2.1. A *constraint system* is a structure $\langle D, \vdash, \mathtt{Var}, \{\exists_{\mathtt{X}} \mid \mathtt{X} \in \mathtt{Var}\}\rangle$ such that

- D is closed under conjunction ($\wedge$); $\vdash \subseteq D \times D$ satisfies:
  - $a \vdash a$; $a \vdash b$ and $b \wedge c \vdash d$ implies that $a \wedge c \vdash d$,
  - $a \wedge b \vdash a$ and $a \wedge b \vdash b$; $a \vdash b$ and $a \vdash c$ implies that $a \vdash b \wedge c$.
- Var is an infinite set of *variables*, such that for each variable $X \in \mathtt{Var}$, $\exists_X : D \to D$ is an operation satisfying usual laws on existentials:
  - $a \vdash \exists_X a$,
  - $\exists_X(a \wedge \exists_X b) \approx \exists_X a \wedge \exists_X b$,
  - $\exists_X \exists_Y a \approx \exists_Y \exists_X a$,
  - $a \vdash b \Rightarrow \exists_X a \vdash \exists_X b$.

A *constraint* is an entailment closed subset of $\mathscr{D}$. For any set of tokens S, we let $\bar{S}$ stand for the set $\{a \in D \mid \exists\{a_1,\ldots,a_k\} \subseteq S \,.\, a_1 \wedge \cdots a_k \vdash a\}$. For any token $a, \bar{a}$ is just the set $\overline{\{a\}}$.

The set of all constraints, denoted $|D|$, is ordered by inclusion, and forms an algebraic lattice with least upper bounds induced by $\wedge$. We will use $\sqcup$ and $\sqcap$ to denote joins and meets of this lattice. We will use $u, v, w, \ldots$ to denote constraints. $\bar{a}$ denotes the embedding of a in $|D|$: $\bar{a} = \{b \in D \mid a \vdash_{\mathscr{D}} b\}$. $\exists$, $\vdash$ lift to operations on constraints.

Of course, in any implementable language, $\vdash$ must be decidable – and as efficient as the intended class of users of the language demand.

Examples of such systems are the system Herbrand (underlying logic programming), FD [11] (finite domains), and `Gentzen` [21].

Example 2.2 (*The Herbrand constraint system*). Let L be a first-order language with equality. The tokens of the constraint system are the atomic and existentially quantified propositions. Entailment can vary depending on the intended use of the predicate symbols but it must include the usual entailment relations that one expects from equality. Thus, for example, $f(X,Y) = f(A, g(B,C))$ must entail $X = A$ and $Y = g(B,C)$.

Example 2.3 (*The FD constraint system*). Its tokens are equalities of variables and expressions saying that the range of a variable is some finite set.

Example 2.4 (*The* `Gentzen` *constraint system*). For timed computation we have found the simple constraint system ($\mathscr{G}$) to be very useful. `Gentzen` provides the very simple level of functionality that is needed to represent signals. The tokens of `Gentzen` are atomic formulas drawn from a pre-specified logical vocabulary; the entailment relation is trivial, i.e. $c_1,\ldots,c_n \vdash_{\mathscr{G}} c$ iff $c = c_i$ for some i.

For the rest of this paper, we assume that we are working within a given constraint system $\langle D, \vdash, \mathtt{Var}, \{\exists_{\mathtt{X}} \mid \mathtt{X} \in \mathtt{Var}\}\rangle$.

3. Causal transition system for cc

3.1. Syntax

The syntax of cc languages is as follows:

$$P ::= a \mid P \parallel P \mid \Box_i a_i \to P_i \mid \exists X \,.\, P \mid g(X) \mid g(X) \mid \{D \,.\, P\}$$
$$D ::= \varepsilon \mid g(X) :: P, D$$

We think of the guarded choice operator $\Box$ as a family of $(2n)$-ary operators that takes n constraints $a_1,\ldots,a_n$ and n processes $P_1,\ldots,P_n$ and builds the process $\Box_i a_i \to P_i$.

3.2. Transition system of cc

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 conjunction of the set of tokens in Γ defined inductively as follows:

$$\sigma(a) = a$$

$$\sigma(P \parallel Q) = \sigma(\exists X\,.\,P) = \sigma(\Box_i a_i \to P_i) = \sigma(g(X)) = \mathtt{true}$$

$$\sigma(\{P_1,\ldots,P_n\}) = \sigma(P1) \wedge \cdots \wedge \sigma(P_n)$$

The transition system relates configurations to configurations and is the least relation satisfying the following axioms and inference rules [24]:

$$\Gamma, (P \parallel Q) \to \Gamma, P, Q \qquad \frac{\sigma(\Gamma) \vdash_{\mathscr{D}} a_i}{\Gamma, (\Box_{i\in I} a_i \to P_i) \to \Gamma,\ P_i}$$

$$\Gamma, \exists X\,.\,P \to \Gamma, P[Y/X] \qquad \Gamma, g(X) \to \Gamma, P(X)$$

$$(Y \text{ new}) \qquad \text{if } g(X) :: P(X) \in D$$

3.3. Observing causality

To detect causality information, we will allow our configurations to remember the set of tokens 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 (where P is an agent, a is a token), satisfying the condition that

$$\text{if } P^a \in \Gamma \quad \text{then } \sigma(\Gamma) \vdash_{\mathscr{C}} a$$

where $\sigma(\Gamma)$ again denotes the conjunction of the set of tokens in Γ defined inductively as follows:

$$\sigma(a^b) = a$$

$$\sigma(P \parallel Q) = \sigma(\exists X\,.\,P) = \sigma(\Box_i a_i \to P_i) = \sigma(g(X)) = \mathtt{true}$$

$$\sigma(\{P_1,\ldots,P_n\}) = \sigma(P1) \wedge \cdots \wedge \sigma(P_n)$$

$\rho(\Gamma)$ denotes the multiset contexted tokens in Γ defined inductively as follows:

$$\rho(\{a^b\}) = \{a^b\}$$

$$\rho(P \parallel Q) = \rho(\exists X\,.\,P) = \rho(\Box_i a_i \to P_i) = \rho(g(X)) = \{\mathtt{true}^{\mathtt{true}}\}$$

$$\rho(\{P_1,\ldots,P_n\}) = \rho(P1) \cup \cdots \cup \rho(P_n)$$

In the rest of this paper, we will identify the agent P with the contexted agent $P^{\mathtt{true}}$. The transition relation may now be defined straightforwardly:

$$\Gamma, (P \parallel Q)^c \to \Gamma, P^c, Q^c \qquad \frac{\sigma(\Gamma) \vdash_{\mathscr{D}} a_i}{\Gamma, (\Box_{i\in I} a_i \to P_i)^c \to \Gamma, P_i^{a_i \wedge c}}$$

$$\Gamma, (\exists X\,.\,P)^c \to \Gamma, P[Y/X]^c \qquad \Gamma, g(X)^c \to \Gamma, P(X)^c$$

$$(Y \text{ new}) \qquad \text{if } g(X) :: P(X) \in D$$

Let $\Gamma \to \Delta$. The causal context of this transition, written $\mathtt{TransCause}(\Gamma \to \Delta)$ is the context of the agents produced in Γ by the transition. In the above transition system, $\mathtt{TransCause}(\cdot)$ is c for all transitions except the case of the guarded choice where it is $a_i \wedge c$.

Example 3.1 (*Example* 1.1: *Contd.*). Here is how the example of two not gates mentioned earlier would be executed under the new transition relation, when started in the presence of constraints $\mathtt{M1}=\mathtt{ok}$, $\mathtt{X1}=0$, $\mathtt{M2}=\mathtt{ok}$, $\mathtt{X2}=1$ (accomplished by $\Uparrow(M1=ok, X1=0, M2=ok, X2=1) = M1=ok^{M1=ok}, X1=0^{X1=0}, M2=ok^{M2=ok}, X2=1^{X2=1}$):

$$\begin{aligned}
&\mathtt{not(M1,X1,Y1)}, \mathtt{not(M2,X2,Y2)}, \Uparrow(\mathtt{M1}=\mathtt{ok}, \mathtt{X1}=0, \mathtt{M2}=\mathtt{ok}, \mathtt{X2}=1)\\
&\quad\to (\mathtt{M1}=\mathtt{ok} \to \mathtt{Y1}=\mathtt{not(X1)} \Box \ldots), \mathtt{not(M2,X2,Y2)},\\
&\qquad \Uparrow(\mathtt{M1}=\mathtt{ok}, \mathtt{X1}=0, \mathtt{M2}=\mathtt{ok}, \mathtt{X2}=1)\\
&\quad\to (\mathtt{Y1}=\mathtt{not(X1)})^{\mathtt{M1}=\mathtt{ok}}, \mathtt{not(M2,X2,Y2)},\\
&\qquad \Uparrow(\mathtt{M1}=\mathtt{ok}, \mathtt{X1}=0, \mathtt{M2}=\mathtt{ok}, \mathtt{X2}=1)\\
&\quad\to (\mathtt{Y1}=\mathtt{not(X1)})^{\mathtt{M1}=\mathtt{ok}}, (\mathtt{M2}=\mathtt{ok} \to \mathtt{Y2}=\mathtt{not(X2)} \Box \ldots),\\
&\qquad \Uparrow(\mathtt{M1}=\mathtt{ok}, \mathtt{X1}=0, \mathtt{M2}=\mathtt{ok}, \mathtt{X2}=1)\\
&\quad\to (\mathtt{Y1}=\mathtt{not(X1)})^{\mathtt{M1}=\mathtt{ok}}, (\mathtt{Y2}=\mathtt{not(X2)})^{\mathtt{M2}=\mathtt{ok}},\\
&\qquad \Uparrow(\mathtt{M1}=\mathtt{ok}, \mathtt{X1}=0, \mathtt{M2}=\mathtt{ok}, \mathtt{X2}=1)
\end{aligned}$$

Now since $\mathtt{Y1}=\mathtt{not(X1)}, \mathtt{X1}=0 \vdash_{\mathscr{D}} \mathtt{Y1}=1$, and $a, b \vdash_{\mathscr{D}} e$ gives $a^c, b^d \vdash e^{c,d}$, the store contains $(\mathtt{Y1}=1)^{\mathtt{M1}=\mathtt{ok},\mathtt{X1}=0}$ and $(\mathtt{Y2}=0)^{\mathtt{M2}=\mathtt{ok},\mathtt{X2}=1}$, which is exactly what we expect as the causal behavior of the program.

3.4. Equivalences of contexted tokens

Execution of a program is thus taken to yield a store of contexted tokens. 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 $a \parallel a \to b$ (in the presence of no other tokens) is different from that obtained from $a \parallel b$, though semantically they should be identical.

Our solution follows from what it means for a program P to produce a contexted token o. If $o = a^b$, it informally means that P, if given input b, would produce a. Similarly if o is the multiset containing o_1, o_2, then it means that P can produce both o_1 and o_2. Thus, contexted tokens themselves can be viewed as finite determinate cc programs, motivating the use of the tools of the semantics of determinate cc to analyze the equivalence of contexted tokens.

We recall the use of closure operators in [12, 24]. A *closure operator* is a function f from constraints to constraints, which is extensive ($f(u) \supseteq u$), idempotent ($f \circ f = f$) and monotone (if $u \supseteq v$ then $f(u) \supseteq f(v)$). An alternative way of presenting a closure operator is as a set of its fixed points, i.e. those constraints v such that $f(v) = v$. We recall that the set of fixed points of a closure operator is a set of constraints 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 $f_A(u) = \sqcap \{v \in A \mid v \supseteq u\}$. In the rest of this paper we will use both these representations interchangeably. Closure operators are ordered pointwise – thus if for all u, $f(u) \subseteq g(u)$, we define $f \leqslant g$. In the set representation, this simply becomes $f \leqslant g$ iff $f \supseteq g$. We will refer to this ordering as the *information ordering*, it is the converse of the usual set ordering.

Definition 3.2. Let M be a multiset of contexted tokens.

$[\![M]\!]$ is defined inductively as follows:

$$[\![\{a^b\}]\!] = \{u \in |D| \mid b \in u \Rightarrow a \in u\}$$

$$[\![\{o_1, \ldots, o_n\}]\!] = [\![o_1]\!] \cap \cdots \cap [\![o_n]\!]$$

$$[\![\exists_X o]\!] = \{u \in |D| \mid \exists \in [\![o]\!], \exists_X u = \exists_X v\}$$

$\sigma(M)$ is defined inductively as follows:

$$\sigma(\{a^b\}) = a$$

$$\sigma(\{o_1, \ldots, o_n\}) = \sigma(o_1) \wedge \cdots \wedge \sigma(o_n)$$

$$\sigma(\exists_X o) = \exists_X(\sigma(o))$$

We are now ready to define equivalences of (multi)sets of contexted tokens.

Definition 3.3. Let M_1, M_2 be two multisets of closure operators. Then, $M_1 \approx M_2$ if $\sigma(M_1) = \sigma(M_2)$ and $[\![M_1]\!] = [\![M_2]\!]$.

3.4.1. A logical justification of Definition 3.3

In this subsection, we identify the relevant entailment relation on contexted tokens that leads to the equivalences $\approx$ on contexted tokens.

Let o, p, q range over contexted tokens. Consider the judgement:

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

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 $o_1,\ldots,o_n$ also satisfies o. In the following, let Γ,Δ range over multisets of contexted tokens. We will allow our contexted tokens to take on a slightly more general syntax: a contexted token will be given by the grammar

$$o ::= a^{true} | o^a | o \wedge o$$

where a is a token; we define $[\![o^a]\!] = \{u \in |D| \| b \in u \Rightarrow a \in [\![o]\!]\}$ and $\sigma(o^a) = \sigma(o)$.

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

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

Regarding the axioms, we just let them follow from the basic entailment relation, making the obvious identification of a^{true} with a. As expected, the Cut rule holds:

$$\frac{a_1,\ldots,a_n \vdash_{\mathscr{D}} b}{a_1^{true},\ldots,a_n^{true} \vdash b^{true}} \qquad \frac{\Gamma \vdash p \quad \Delta, p \vdash o}{\Gamma, \Delta \vdash o}$$

Now assume (1) that a process P satisfies all the assertions in Γ together with p^a, (2) if any process P satisfies all the assertions in Γ, it satisfies a, and that (3) for any process it is the case that if it satisfies Γ, and p, 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 p^a; therefore it must be the case that P can, on its own, procedure p, and hence P satisfies p. But then, by (3) P satisfies o. This leads to the validity of the inference rule:

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

Now, it remains to consider the conditions under which it can be established that any program P satisfying Γ must satisfy o^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 o. 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 o. But if Q,a can produce o, then it must be the case that when Q is supplied a by the environment it can produce o. Hence we have

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

We have the obvious rules for conjunction:

$$\frac{\Gamma, p, q \vdash o}{\Gamma, p \wedge q \vdash o} \qquad \frac{\Gamma \vdash o \quad \Gamma \vdash o'}{\Gamma \vdash o \wedge o'}$$

Thus, the logic of contexted tokens turns out to be intuitionist logic (over the underlying constraint system). In fact we can show the following theorem:

Theorem 3.4. *Let o be a contexted token and Γ be a set of contexted tokens. Then $[\![\Gamma]\!] \subseteq [\![o]\!]$ iff $\Gamma \vdash o$.*

Proof. The "if" part follows from the discussion above, and the definition of $[\![\cdot]\!]$ described earlier.

Let $[\![\Gamma]\!] \subseteq [\![o]\!]$. We use induction on the structure of o. If $o = o_1 \wedge o_2$, it follows that $[\![\Gamma]\!] \subseteq [\![o_1]\!]$ and $[\![\Gamma]\!] \subseteq [\![o_2]\!]$, so by induction and the right conjunction rule, we have $\Gamma \vdash o$. If $o = o_1^a$, then $[\![\Gamma, a]\!] \subseteq [\![o_1]\!]$ by the definition of closure operators, so we can use induction and the right conjunction rule to prove $\Gamma \vdash o$.

Now suppose $o = a^{true}$. We will use induction on the number n of implications (including nested ones) in Γ of the form p^d, $d \neq true$. Let $\sigma(\Gamma) = \{c \mid c^{true} \in \Gamma\}$, and $\sigma(\Gamma)^{true} = \{c^{true} \mid c^{true} \in \Gamma\}$.

If $n = 0$, then we can use the entailment rule to get the result, from the fact that $[\![\sigma(\Gamma)^{true}]\!] \subseteq [\![a^{true}]\!]$ iff $\sigma(\Gamma) \vdash_{\mathscr{D}} a$.

Let $n > 0$. If $\sigma(\Gamma) \vdash_{\mathscr{D}} a$, we can apply weakening and eventually the entailment rule, and we are done. Otherwise, there must be some $p^d \in \Gamma$, where $\sigma(\Gamma) \vdash_{\mathscr{D}} d$. If this were not the case then $\sqcup\sigma(\Gamma) \in [\![\Gamma]\!]$, but $\sqcup\sigma(\Gamma) \notin [\![a^{true}]\!]$ as $\sigma(\Gamma) \not\vdash_{\mathscr{D}} a$. Now, since $[\![\sigma(\Gamma)^{true}, p^d]\!] = [\![\sigma(\Gamma)^{true}, p]\!]$, we know that $[\![\Gamma, p]\!] \subseteq [\![a^{true}]\!]$. Thus by the induction hypothesis, $\Gamma, p \vdash a^{true}$. Since $\sigma(\Gamma) \vdash_{\mathscr{D}} d$, we can prove $\Gamma \vdash d^{true}$ by weakening and entailment and we have the result by using the left implication rule.

The $[\![\cdot]\!]$ semantics is also sound (in the above sense) for the rules for existential quantification. However, as in cc, intuitionistic logic makes more distinctions than are possible to make in cc.[2] Consider the multiset $M = \{\exists_X \texttt{false}^{c_1}, \exists_X.\texttt{false}^{c_2}\}$ of contexted tokens:: $d \in [\![M]\!]$ iff d is `false` or $\exists_X d$ does not have as much information as c_1 or c_2. However, this is exactly $[\![N]\!]$, where $N = \{\exists_X.(\texttt{false}^{c_1}, \texttt{false}^{c_2})\}$. Hence, $[\![A]\!] = [\![B]\!]$. However, it is not the case that $M \vdash N$ in the above logic with added rules for existentials. Despite this mismatch with intuitionist logic, we choose to go with Definition 3.3 as our notion of equivalence of contexted tokens because of the strong computational evidence supporting the closure operator view provided by the full abstraction theorems for determinate cc languages [12, 24]. □

[2] This discussion is based on the following example for cc. Consider the program A:

$$\exists X.(c_1 \rightarrow \texttt{false}) \,\|\, \exists X.(c_2 \rightarrow \texttt{false}) \tag{6}$$

d is a quiescent point of this program iff d is `false` or $\exists_X d$ does not have as much information as c_1 or c_2. However, this is exactly the behavior of B:

$$\exists X.(c_1 \rightarrow \texttt{false} \,\|\, c_2 \rightarrow \texttt{false}) \tag{7}$$

However it is not the case that $A \vdash B$ in either intuitionist or classical logic.

3.5. Properties of the causal transition system

The properties of the standard cc transition relation life over to the causal transition system.

Lemma 3.5. *The transition relation* $\rightarrow$ *satisfies*

- *Monotonicity*: $A \rightarrow A' \Rightarrow A,\ B \rightarrow A', B$.
- *Extensivity*: $A \rightarrow A' \Rightarrow \sigma(A') \supseteq \sigma(A),\ \rho(A') \supseteq \rho(A)$.
- *Idempotence*: *If* $A \rightarrow A' \nrightarrow \Rightarrow$, *then* $A, \exists_{\vec{\mathbf{Y}}} \rho(A') \rightarrow A' \nrightarrow$ *where* $\vec{\mathbf{Y}}$ *are the variables introduced in the derivation.*

Note that extensivity implies that if $A \rightarrow A'$, then $[\![\rho(A')]\!] \subseteq [\![\rho(A)]\!]$, i.e. the closure operator associated with the contexted store increases along a derivation.

We next explore how some syntactic classes of equivalences on contexted tokens do not affect the transition relation.

Definition 3.6. Let M be a multiset of contexted tokens. Define $\bar{M} = \{a^b \mid [\![M]\!] \subseteq [\![o]\!], \sigma(M) \vdash_{\mathscr{D}} a \wedge b\}$.

We can immediately see that $M \subseteq \bar{M},\ \bar{M} = \bar{\bar{M}},\ M \subseteq N \Rightarrow \bar{M} \subseteq \bar{N}$. Thus if $a^b \in \bar{M}$, then $a^a \in \bar{M}$ and $a_1^{b_1} \in \bar{M}$ where $a \vdash_{\mathscr{D}} a_1,\ b_1 \vdash_{\mathscr{D}} b,\ \sigma(M) \vdash_{\mathscr{D}} b_1$. A simple calculation shows

- $\{a^b\} \approx \{a^b, a^a\} \approx \{a^b, a_1^b\}$ where $a \vdash_{\mathscr{D}} a_1,\ b_1 \vdash_{\mathscr{D}} b$.
- $\{a_1^b, a_2^b\} \approx \{(a_1 \wedge a_2)^b\}$.

Thus, $M \approx \bar{M}$. The following lemma, proved by a simple inspection of the transition rules, shows that the transition relation is insensitive to the distinction between M and $\bar{M}$.

Lemma 3.7. *Let* $o' \in \bar{o}$. *Then*

$$\Gamma, o \rightarrow \Gamma' \Leftrightarrow \Gamma, o, o' \rightarrow \Gamma', o'$$

In the rest of this subsection, for technical convenience, we work with a variant of the transition relation that maintains its multiset of contexted constraints (given by the $\rho(\cdot)$ function), in a form that is closed under the $\overline{(\cdot)}$ operation. This assumption is justified by Lemma 3.7.

The transition system permits some kinds of permutation of derivations.

Lemma 3.8. *Let* $\Gamma_1 \rightarrow \Gamma_{11} \rightarrow \Gamma_2$ *such that* $\sigma(\Gamma_1) \vdash \texttt{TransCause}(\Gamma_{11} \rightarrow \Gamma_2)$ *and the agent transformed by* $\Gamma_{11} \rightarrow \Gamma_2$ *was not produced by the transition* $\Gamma_1 \rightarrow \Gamma_{11}$. *Then, there is a* Δ *such that* $\Gamma_1 \rightarrow \Delta \rightarrow \Gamma_2$ *and the agent transformed in* $\Gamma_1 \rightarrow \Delta$ *is the same as was transformed in* $\Gamma_{11} \rightarrow \Gamma_2$.

Proof. Since the primary source agent (the one that is changed in the transition) of the transition $\Gamma_{11} \rightarrow \Gamma_2$ was not produced in the previous transition, it is present in Γ.

Hence, this transition is enabled in Γ and is not in conflict with the transition $\Gamma_1 \to \Gamma_{11}$ (a transition can be in conflict with another iff they lie along different branches in a choice, which is clearly not the case here as both occurred in the same derivation). Hence the result. □

Note that if $\texttt{TransCause}(\Gamma_{11} \to \Gamma_2) \not\vdash \texttt{TransCause}(\Gamma_1 \to \Gamma_{11})$, then the primary source agent of $\Gamma_{11} \to \Gamma_2$ could not have been produced by $\Gamma_1 \to \Gamma_{11}$ as causes are cumulative, so the above lemma would be applicable. Thus, given a sequence of derivations form Γ_1 to Γ_2, it is possible to rearrange the derivations so that the subderivations with causes $\leqslant a$ is constructed first.

Corollary 3.9. *Let $\Gamma_1 \to^* \Gamma_2$ and a such that $\sigma(\Gamma_1) \vdash_{\mathscr{D}} a$. Then, there is a derivation $\Gamma_1 \to^* \Gamma_2$ such that transitions with causes smaller than a are executed earlier: formally, if $t_1, t_2, \ldots, t_n$ is the sequence of transitions in $\Gamma_1 \to^* \Gamma_2$, then $i \leqslant j$, $a \vdash_{\mathscr{D}} \texttt{TransCause}(\texttt{t}_\texttt{j}) \Rightarrow a \vdash_{\mathscr{D}} \texttt{TransCause}(\texttt{t}_\texttt{i})$.*

Proof. Let t_i be the first transition with cause smaller than a. Then $\texttt{TransCause}(\texttt{t}_\texttt{i}) \not\vdash \texttt{TransCause}(\texttt{t}_\texttt{j})$ for any $j < i$. So t_i can be permuted to the beginning by Lemma 3.8. Repeat this process till we are done. □

We now attempt to extract the semantic content of the above operational analysis of causality. If a program produces a contexted token a^b, then providing it b suffices to create a.

Lemma 3.10. *If $\Gamma,\ a^a \to \Gamma',\ a^b, a^a$, then $\Gamma,\ b^b \to \Gamma', a^b, b^b$.*

Proof. The key case of the proof is when $a^b \notin \rho(\Gamma)$. In this case, the transition is the reduction of a subprogram of the form $b \to \ldots$. A simple inspection of the transition rule yields the required proof. □

The input always causes all the outputs.

Lemma 3.11. *If $P,\ a^a \to^* Q$, then $[\![\exists_{\vec{\mathbf{X}}}.\rho(Q)]\!](a) = \exists_{\vec{\mathbf{X}}}.\sigma(Q)$, where $\vec{\mathbf{X}}$ are the variables introduced in the derivation.*

Proof. Induction on the length of the derivation using Lemma 3.10. □

Lemma 3.12. *Let $P, a^a \to \Gamma_1 \to \Gamma_2 \to \cdots$ be any derivation T, finite or infinite (we assume a is a possibly infinite set of tokens). Let its output be $o = \exists_{\vec{\mathbf{X}}} \bigcup_i \rho(\Gamma_i)$, where $\vec{\mathbf{X}}$ are the new variables introduced in the derivation. If $a \subseteq [\![o]\!](b)$, then there is a derivation $P, b^b \to \cdots$ of the same length as the original derivation T, such that its output o' satisfies $[\![o]\!] = [\![o']\!]$.*

Proof. $a \subseteq [\![o]\!](b)$ means that there is a sequence of contexted tokens $o_i \in o$, $o_i = c_i^{d_i}$, such that $b, \sigma(P), c_1, \ldots, c_n \vdash_{\mathscr{D}} d_{n+1}$, and $b, \sigma(P), c_1, \ldots \vdash_{\mathscr{D}} a$, where the last entailment

means that each token in a is entailed by a finite subset of the tokens on the left.[3]

We will inductively build up a derivation starting from P, b^b such that after stage i, the token o_i is in the contexted store of the configuration reached so far. Then it will be clear that if the output of the derivation is o', then o' will have exactly the same contexted tokens as o, completing the proof. □

Consider the finite prefix of T whose last transition is the one that adds o_1 to the store. Using Corollary 3.9 we can move all transitions with causes less than $\sigma(P) \wedge b$ to the beginning of the prefix. Thus, we get a permutation of T, $T_1 - P$, $a^a \rightarrow^* \Delta \rightarrow \ldots$ such that $o_1 \in \rho(\Delta)$. This is possible as the cause of o_1 is entailed by $\sigma(P) \wedge b$. Note that since all the causes of the transitions reaching Δ were less than $b \wedge \sigma(P)$, we have a derivation $P, b^b \rightarrow^* \Delta'$, where Δ' is Δ with a^a replaced by b^b.

Now similarly we can consider the finite prefix of T_1 which adds o_2 to the store. If it occurs before Δ then $o_2 \in \rho\Delta$ and there is nothing to permute or add to the derivation from P, b^b. Otherwise we repeat the same process, repeatedly increasing the store. Since the contexted tokens produced in this derivation are all the o_i's which together entail a, we can conclude that $o \approx o'$ (note that if there are some contexted tokens in o that were not in o_i, these will still be added to the store, as each of their causes is entailed by a finite number of tokens from a along with $\sigma(P)$, and this finite information is added at some finite stage at which point the transitions adding these tokens will be enabled, and hence done). Note that the input tokens a^a, b^b are merely variants of `true`, and play no part in this equivalence.

4. The causal may semantics

This style of causal semantics of cc programs allows us to observe what *may* be true of terminating runs of the system. Recall that the observation of a terminating run is the *contexted store* $\rho(\Gamma)$ of the terminating configuration Γ.

4.1. Notation

In order to accommodate hiding, our observations will need to hide all the new variables introduced in the transitions. Thus, they will be of the form $\exists_V.o$, where V is the set of new variables introduced in the derivation. For the rest of this paper, we will always use V to stand for this set for the derivation under consideration.

We also make the following assumptions about procedures and procedure calls to make the notation simple. We assume that procedures have only one parameter. Secondly, we assume that each procedure g has a variable X_g associated with it that is

[3] This can be intuitively seen from the fact that if we regard o as a possibly infinite determinate cc program, then $a \subseteq [\![o]\!](b)$ means that there is a cc derivation from o, b with output at least a. Since all the tokens in o are of the form c^d, each transition in this derivation will be of the form $O, c^d \rightarrow c$, if $\sigma(O) \vdash_{\mathscr{D}} d$. Thus each transition can be uniquely identified with a contexted token $o_i \in o$, $i = 1, 2, \ldots$.

used only in the body of the procedure. We write the unique declaration of a procedure $g(X):A$ in the form: $g(X):\mu g.C[g]$ to indicate that the body of g is a context that can use g. In this notation a call $g(Y)$ is treated as $\exists X_g.[X_g = Y \parallel C[g]]$.

The definition of the may operational semantics uses the notion of *size* of a contexted store. Intuitively, the size of a contexted store o, denoted $\|o\|$, is the lub of all the tokens occurring in it.

Definition 4.1. $\|o\|$ is defined inductively as follows:

$$\|a\| = a, \quad \|a^b\| = a \sqcup b, \quad \|o \wedge o'\| = \|o\| \sqcup \|o'\|, \quad \|\exists_X o\| = \exists_X \|o\|$$

Definition 4.2. For a collection of tokens $c_1,\ldots,c_n$, define $\Uparrow(c_1,\ldots,c_n)$ to be the set of contexted tokens $c_1^{c_1},\ldots,c_n^{c_n}$.

The execution of the program P with input tokens $c_1,\ldots,c_n$ is to be thought of as the execution of the agent $\mathtt{P},\Uparrow(\mathtt{c_1},\ldots,\mathtt{c_n})$.

4.2. Operational semantics

o is an observation of a program P in context c if

$$P,\Uparrow(c) \rightarrow^* \Gamma, \quad o = \exists_V.\rho(\Gamma)$$

The *may operational semantics* is given as follows – intuitively, the may operational semantics of a program P (viewed as a contexted program $P^{\mathtt{true}}$) is a collection of the causality observations (i.e. contexted stores) of all possible *terminated* computations of P.

Definition 4.3. The operational semantics is given by

$$\mathcal{O}_H[\![P]\!] = \{o \mid \exists o', \|o'\| = \|o\|, P,\Uparrow(\|o\|) \rightarrow^* \Gamma \not\rightarrow, o' = \exists_V.\rho(\Gamma), [\![o]\!] \supseteq [\![o']\!]\}$$

The above definition builds downclosed sets of observations. Thus, we note that the above definition allows us to identify two sets of observations which have the same maximal elements, giving us the may tests of [8].

4.3. Denotational semantics

4.3.1. Bounded closure operators

As defined earlier, a *closure operator* f can be represented as a set of constraints closed under greatest lower bounds. We will use the cc operations on closure operators– thus $a \rightarrow f = \{u \in |D| \mid a \in u \Rightarrow u \in f\}$, and $\exists_X f = \{u \in |D| \mid \exists v \in f, \exists_X u = \exists_X v\}$. As before, closure operators are ordered by the converse of the usual set ordering, least upper bounds in this ordering turn out to be set intersection.

Definition 4.4. A *bounded closure operator* (bco) is a pair (f,u), where f is a closure operator, and u is a constraint, $u \in f$.

The u determines the *domain* of the bco – this is defined as $u\downarrow$, the set of constraints smaller than u. The set of elements bigger than u is written $u\uparrow$. Thus if $(f,u),(g,u)$ are bco's, with $f\cap u\downarrow = g\cap u\downarrow$, then we consider $(f,u)=(g,u)$. The equivalence classes of bco's under this equality are closed under glb's – thus if $(f,u)=(g,u)$ then $(f,u)=(f\sqcap g,u)$, and this is true for infinite glb's as well. The ordering on closure operators is extended to bco's – $(f,u)\leqslant(g,u)$ iff $f\supseteq g\cap u\downarrow$. 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 partial order closed under lubs of directed sets, called **Obs**. The carrier set of this domain is denoted as $|\mathbf{Obs}|$. The compact elements of **Obs** are of the form (f,u) where f is a compact closure operator – compact closure operators are generated by the grammar $f ::= a \mid a\to f \mid f\wedge f$. The set of all compact elements of **Obs** is denoted **CObs**.

4.3.2. A powerdomain

In the rest of this section we define the semantics of cc languages by examining the relational powerdomain constructions on this domain [17]. In this paper, are technically using the powerdomain construction with the empty set, rather than the more traditional ones without the empty set. Furthermore, we use the representation theorems to simplify the presentation of the powerdomain constructions (see p. 90 of [17]).

Definition 4.5. The elements of the relational powerdomain on **Obs** are sets S of compact bco's satisfying the closure condition

$$(f,u)\in S, (g\cap u\downarrow)\supseteq(f\cap u\downarrow)\Rightarrow(g,u)\in S$$

The ordering relation is given by subset inclusion: $S_1\sqsubseteq S_2\Leftrightarrow S_1\subseteq S_2$.

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

4.3.3. Denotational semantics

The semantics of the various program combinators is given as follows:

$$\mathscr{H}[\![a]\!] \stackrel{d}{=} \{(f,u)\in\mathbf{CObs} \mid a\in u, f\supseteq \bar{a}\uparrow\cap u\downarrow\}$$

$$\mathscr{H}[\![P \parallel Q]\!] \stackrel{d}{=} \{(h,u)\in\mathbf{CObs} \mid \exists(f,u)\in\mathscr{H}[\![P]\!],(g,u)\in\mathscr{H}[\![Q]\!].\ h\supseteq f\cap g\cap u\downarrow\}$$

$$\mathscr{H}[\![\Box_{i\in I} a_i\to P_i]\!] \stackrel{d}{=} \{(u\downarrow,u)\in\mathbf{CObs} \mid \forall i\in I. a_i\notin u\} \cup \bigcup_{i\in I}\{(f,u)\in\mathbf{CObs} \mid a_i\in u, \exists(f_i,u)\in\mathscr{H}[\![P_i]\!], f\supseteq(a_i\to f_i)\cap u\downarrow\}$$

$$\mathscr{H}[\![\exists X\,.\,P]\!] \stackrel{d}{=} \{(f,u) \in \mathbf{CObs} \mid \exists (h,u) = (f,u), \exists (g,v) \in \mathscr{H}[\![P]\!].$$
$$\exists X\,.\,g = \exists X\,.\,h, \exists_X u = \exists_X v, g(\exists_X v) = v\}$$
$$\mathscr{H}[\![g(Y)]\!] \stackrel{d}{=} \mathscr{H}[\![\exists X_g.[X_g = Y \parallel g]]\!]$$
$$\mathscr{H}[\![g]\!] \stackrel{d}{=} \mu g.\mathscr{H}[\![C[g]]\!], \quad \text{where } g(X)\text{: } C[g] \text{ is the procedure declaration}$$

All the above operations on processes are monotone and continuous with respect to the ordering on sets of processes. Thus, recursion is treated via least fixed points, written μ above.

We draw the reader's attention to the use of the bound (in a bounded closure operator) to resolve the choice in the guarded choice operator. Note the extra clause in the definition of $\exists X\,.\,P$, which states that $g(\exists_X v) = g(v)$. 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 $\exists_X v$ as on v.

A few examples will illustrate the nature of the denotational semantics. The first example shows a characteristic feature of relational powerdomain style semantics – adding more branches to a process moves it up the ordering in the powerdomain.

Example 4.6. Let P_i be a collection of processes indexed by i. Let a be a token. Then $\mathscr{H}[\![a \to P_i]\!] \sqsubseteq \mathscr{H}[\![\Box_{i\in I}\, a \to P_i]\!]$. We justify the inequation by noting that the sematic interpretation of bounded choice in the special case when all the guards are identical is just union.

The next couple of examples indicate the treatment of termination by the semantics – only terminating runs are counted.

Example 4.7. Let the process P_1 be defined recursively as $P_1 :: P_1$. Then the $\mathscr{H}[\![P_1]\!] = \bigcup_i \mathscr{H}[\![C^i(\emptyset)]\!]$, where $C(X) = X$. Thus $C^i(\emptyset) = \emptyset$, so $\mathscr{H}[\![P_1]\!] = \emptyset$.

Example 4.8. Let the process P_2 be defined recursively as $P_2 :: b \parallel P_2$. Then $\mathscr{H}[\![P_1]\!] = \bigcup_i \mathscr{H}[\![C^i(\emptyset)]\!]$, where $C(X) = b \parallel X$. Thus $C(\emptyset) = \emptyset$, so $\mathscr{H}[\![P_2]\!] = \emptyset$.

The next example further clarifies termination issues in the relational semantics – non-terminating runs are ignored.

Example 4.9. Let P be any process. Consider the processes $P_3 = a \to P \,\Box\, a \to P_1$, and $P_4 = a \to P \,\Box\, a \to P_2$, where P_1, P_2 are as above. Then $\mathscr{H}[\![P_3]\!] = \mathscr{H}[\![P_4]\!] = \mathscr{H}[\![a \to P]\!]$.

Example 4.10. The operation of adding constraints is idempotent – a key ingredient of the cc paradigm

$$\mathscr{H}[\![a]\!] = \mathscr{H}[\![a \parallel a]\!]$$

The next example shows another characteristic feature of the may style semantics – a "downward closure" property.

Example 4.11. Let $P = \mathtt{true} \to a \,[]\, true \to b \to b$ and $Q = P \,[]\, \mathtt{true} \to b \to a$, where $a \vdash b$. Then P and Q are indistinguishable with respect to the operational semantics, as the extra output of Q, i.e. a^b is less than the output $a^{\mathtt{true}}$ of P, and we observe only maximal outputs.

Example 4.12. The following equational laws hold:

$$\mathscr{H}[\![a \,\|\, P]\!] = \mathscr{H}[\![a \,\|\, a \to P]\!]$$
$$\mathscr{H}[\![b \to (A \,\|\, B)]\!] = \mathscr{H}[\![b \to A \,\|\, b \to B]\!]$$

A sketch of the proof of the forward direction of the first identity is as follows. Let $(f,u) \in \mathscr{H}[\![a \,\|\, P]\!]$. Then $f \supseteq g \cap h \cap u\!\downarrow$, where $(g,u) \in \mathscr{H}[\![a]\!]$ and $(h,u) \in \mathscr{H}[\![P]\!]$. Thus $g \supseteq \bar{a}\!\uparrow \cap u\!\downarrow$, and $\bar{a}\!\uparrow \cap h = \bar{a}\!\uparrow \cap a \to h$. Now $(\bar{a}\!\uparrow \cap a \to h, u) \in \mathscr{H}[\![a \,\|\, a \to P]\!]$, since $a \in u$. Thus $f \supseteq g \cap h \cap u\!\downarrow \supseteq u\!\downarrow \cap \bar{a}\!\uparrow \cap a \to h$, so $(f,u) = \mathscr{H}[\![a \,\|\, a \to P]\!]$.

Example 4.13. Let X not be free in b, A. Then,

$$\mathscr{H}[\![\exists X \,.\, b \to X \,\|\, X \to B]\!] = \mathscr{H}[\![b \to B]\!]$$
$$\mathscr{H}[\![\exists X \,.\, (A \,\|\, B)]\!] = \mathscr{H}[\![A \,\|\, \exists X \,.\, B]\!]$$

These laws suffice to prove the Law of Intermediate Causation (Eq. (5)). The proofs are straightforward and hence omitted.

4.4. Full abstraction

We now show that the denotational semantics is fully abstract with respect to the operational semantics. We can represent an observation o as a pair $([\![o]\!], \|o\|)$ – then $\mathscr{O}_H[\![P]\!]$ can be regarded as a set of bco's

Theorem 4.14.

$$\mathscr{O}_H[\![a]\!] = \mathscr{H}[\![a]\!]$$
$$\mathscr{O}_H[\![P \,\|\, Q]\!] = \{(h,u) \in \mathbf{CObs} \mid \exists (f,u) \in \mathscr{O}_H[\![P]\!], (g,u) \in \mathscr{O}_H[\![Q]\!]. \; h \supseteq f \cap g \cap u\!\downarrow\}$$
$$\mathscr{O}_H[\![[]_{i\in I} a_i \to P_i]\!] = \{(\downarrow u, u) \in \mathbf{CObs} \mid \forall i \in I. a_i \notin i\} \cup \bigcup_{i\in I} \{(f,u) \in \mathbf{CObs} \mid a_i \in u, \exists (f',u) \in \mathscr{O}_H[\![P_i]\!], f \supseteq (a_i \to f') \cap u\!\downarrow\}$$
$$\mathscr{O}_H[\![\exists X \,.\, P]\!] = \{(f,u) \in \mathbf{CObs} \mid \exists (h,u) = (f,u). \exists (g,v) \in \mathscr{O}_H[\![P]\!].$$

$$\exists_X u = \exists_X v, g(\exists_X v) = v, \exists X \,.\, g = \exists X \,.\, h\}$$

$$\mathcal{O}_H[\![\mu X \,.\, C[X]]\!] = \textit{the least fixed point of } \mathcal{O}_H[\![C[]]\!]$$

Proof.

- Since $a, u^u \nrightarrow, (f,u) \in \mathcal{O}_H[\![a]\!]$ iff $f \supseteq [\![a, u^u]\!] \cap u\downarrow = a\uparrow \cap u\downarrow$, i.e. $(f,u) \in \mathscr{H}[\![a]\!]$.
- If $(f,u) \in \mathcal{O}_H[\![P \parallel Q]\!]$, then there is an observation o of $P \parallel Q$ such that $(f,u) \subseteq ([\![o]\!], \|o\|)$. Let the derivation of o be $P,\ Q,\ u^u \rightarrow^* R \nrightarrow$. By Lemma 3.8, we can permute the transitions so that all the transitions involving P and agents derived from it are done before any transitions involving Q. Thus we have $P \parallel Q,\ u^u \rightarrow P,\ Q,\ u^u \rightarrow^* P',\ Q,\ u^u \rightarrow^* P',\ Q',\ u^u \nrightarrow$, and $P', Q' = R$. Let $o_1 = \exists_{\vec{\mathbf{Y}}_1} \sigma(P', u^u)$ and $o_2 = \exists_{\vec{\mathbf{Y}}_2} \sigma(Q', u^u)$, where Y_1 is the set of new variables introduced in $P, Q, u^u \rightarrow^* P', Q, u^u$ and Y_2 is the set introduced in $P', Q, u^u \rightarrow^* P', Q', u^u$. Then o_1 is an observation of P and o_2 is an observation of Q, and $o = o_1 \wedge o_2$. Thus $([\![o_1]\!], u) \in \mathcal{O}_H[\![P]\!]$ and $([\![o_2]\!], u) \in \mathcal{O}_H[\![Q]\!]$ and $f \supseteq ([\![o_1]\!] \cap [\![o_2]\!]) \cap u\downarrow$.

  Conversely, if $(g,u) \in \mathcal{O}_H[\![P]\!]$ and $(h,u) \in \mathcal{O}_H[\![Q]\!]$, suppose o_1 and o_2 are the observations of P and Q, with $(g,u) \supseteq ([\![o_1]\!], \|o_1\|)$ and $(h,u) \supseteq ([\![o_2]\!], \|o_2\|)$, thus $\|o_1\| = \|o_2\|$. Then by monotonicity we get $o_1 \wedge o_2$ as an observation of $P \parallel Q$. Now since $(g \cap h, u) \supseteq ([\![o_1 \wedge o_2]\!], \|o_1 \wedge o_2\|)$, we have $(f,u) \in \mathcal{O}_H[\![P \parallel Q]\!]$ for any $(f,u) \supseteq (g \cap h, u)$.
- Let $(f,u) \in \mathcal{O}_H[\![\square_{i\in I} a_i \rightarrow P_i]\!]$. Let o be a observation of $\square_{i\in I} a_i \rightarrow P_i$, with $(f,u) \supseteq ([\![o]\!], \|o\|)$. If $\forall i \in I . a_i \notin u$, then $o = \mathtt{true}$, so $[\![o]\!] = \downarrow u$, so (f,u) is in the RHS. Otherwise, $\square_{i\in I} a_i \rightarrow P_i,\ u^u \rightarrow P_i^{a_i},\ u^u$ while making the observation o. Thus $a_i \in u$. Also, $o = o'^{a_i} \wedge u^u$, where $o',\ u^u$ is an observation of P_i. Thus $([\![o']\!], u) \in \mathcal{O}_H[\![P_i]\!]$, and $f \supseteq [\![a_i \rightarrow o']\!] \cap u\downarrow$, so (f,u) is in the RHS.

  The converse follows by reversing the above arguments.
- Let $(f,u) \in \mathcal{O}_H[\![\exists X \,.\, P]\!]$ such that o is an observation of $\exists X \,.\, P$ satisfying $(f,u) \supseteq ([\![o]\!], \|o\|)$. Thus means that there is a derivation $\exists X \,.\, P,\ u^u \rightarrow P[Y/X],\ u^u \rightarrow \cdots$ such that
  - Y is new in u and P,
  - $o = \exists_X o',\ u^u$, where o' is the observation of the derivation $P, \Uparrow (\exists_X u) \rightarrow \cdots$ corresponding to the above derivation $P[Y/X],\ u^u \rightarrow \cdots$.

  From Lemma 3.11 it follows that $o'(\exists_X \|o'\|) \supseteq o'(\exists_X u) = \|o'\|$. Furthermore, since $\exists_X u \subseteq \|o'\|$, we have $\exists_X o' = \exists_X o' \sqcup \exists_X u^u = \exists_X o'$.

  Conversely, let $(g,v) \in \mathcal{O}_H[\![P]\!]$, with $g(\exists_X v) = v$. Let (f,u) be such that $\exists X \,.\, g = \exists X \,.\, h \exists_X u = \exists_X v, (h,u) = (f,u)$. We will show that $(f,u) \in \mathcal{O}_H[\![\exists X \,.\, P]\!]$.

  Since $(g,v) \in \mathcal{O}_H[\![P]\!]$, we deduce that there is a derivation starting at $P, v^v \rightarrow \cdots \rightarrow Q \nrightarrow$, such that $\sigma(Q) = o, g \supseteq [\![o]\!]$. Since, $g(\exists_X v) = v$, we have $[\![o]\!](\exists_X v) = v$; using Lemma 3.12, we deduce that $P,\ (\exists_X v)^{\exists_X v} \rightarrow \cdots \rightarrow Q$. This sequence of reductions can be mimicked by $\exists X \,.\, P, u$, and the result follows.
- If $P = \mu X \,.\, C[X]$, then we want to show that $\mathcal{O}_H[\![P]\!]$ is the least fixed point of $\mathcal{O}_H[\![C[]]\!]$, which is an operator from processes to processes. By the above proofs, it follows that $C[]$ is a continuous operator, thus it has a least fixed point. Also by the rules

given above, we can show that

$$\mathscr{O}_H[\![C[\,]]\!](\mathscr{O}_H[\![Q]\!]) = \mathscr{O}_H[\![C[Q]]\!]$$

for any program Q. It follows that the least fixed point of $\mathscr{O}_H[\![C[\,]]\!]$ is $\bigcup_i \mathscr{O}_H[\![C^i[\emptyset]]\!]$, where $\emptyset$ is the empty process (the least element of the powerdomain), and $C^i[\emptyset] = C[C^{i-1}[\emptyset]]$.

Now Since every $(f,u) \in \mathscr{O}_H[\![P]\!]$ is obtained by a finite derivation, the rule for recursion is applied at most finitely many times in each derivation, so $(f,u) \in \mathscr{O}_H[\![C^i(\phi)]\!]$ for some i. Conversely, any $(f,u) \in \mathscr{O}_H[\![C^i(\phi)]\!]$ can be obtained from P by unrolling the recursion i times. □

A structural induction now yields the full abstraction theorem.

Corollary 4.15. *If P, Q are two indeterminate programs, then $\mathscr{H}[\![P]\!] = \mathscr{H}[\![Q]\!]$ iff $\mathscr{O}_H[\![P]\!] = \mathscr{O}_H[\![Q]\!]$.*

4.5. Logical form

We now present the relational semantics in a logical form in the spirit of domain theory in logical form [1]. 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. Due to the standard mismatch between hiding and existential quantification, we will treat only programs without hiding, and come back to hiding at the end of this section.

Intuitively, the relational semantics of a program consists of all the properties which are satisfied exactly by some execution sequence of the program. Properties are generated by the following syntax:

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

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

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

$$\frac{a_1, \ldots, a_n \vdash_{\mathscr{C}} b \quad \forall i.\ a_i \in u}{a_1, \ldots, a_n \Vdash_u b}$$

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

$$\frac{\sigma(\Gamma) \vdash_{\mathscr{C}} a_i \quad \Gamma, A_i \Vdash_u \phi}{\Gamma, \Box_{i \in I} a_i \to A_i \Vdash_u \phi} \qquad \frac{\Gamma, a \Vdash_u \phi \quad a \in u}{\Gamma \Vdash_u a \to \phi}$$

$$\frac{\forall i \in I. a_i \notin u \quad \Gamma \Vdash_u \phi}{\Gamma, \Box_{i \in I} a_i \to A_i \Vdash_u \phi} \qquad \frac{\Gamma, A(X) \Vdash_u \phi \quad g(X) :: A(X)}{\Gamma, g(X) \Vdash_u \phi}$$

We draw the reader's attention to the rule for guarded choice. The rule ensures that *some* enabled branch of the guarded choice satisfy the property, a characteristic feature of may testing style semantics.

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 4.16. *For any hiding free program $P, P \Vdash_u \phi$ if and only if $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket P\rrbracket$.*

Proof. Let $P \Vdash_u \phi$. We will show that $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket P\rrbracket$. This will be shown by induction on the proof tree for $P \Vdash_u \phi$:

- Clearly if $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket \Gamma, A, B, \Delta\rrbracket$ then $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket \Gamma, B, A, \Delta\rrbracket$, since the two denotations are the same.
- $\phi \vdash_{IL} \psi$ means $\llbracket\phi\rrbracket \subseteq \llbracket\psi\rrbracket$. So $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket\Gamma\rrbracket$ means $(\llbracket\psi\rrbracket, u) \in \mathscr{H}\llbracket\Gamma\rrbracket$ as all processes are upwards closed.
- $a_1, \ldots, a_n \vdash_{\mathscr{C}} b$ means that $\llbracket b\rrbracket \supseteq a_1 \sqcup \cdots \sqcup a_n\uparrow$. Since for all i $a_i \in u$, $a_1 \sqcup \cdots \sqcup a_n \in u$ so $(\llbracket b\rrbracket, u) \in \mathscr{H}\llbracket a_1 \sqcup \cdots \sqcup a_n\rrbracket$.
- The property holds for the left parallel rule trivially. For the right rule, we note that $\llbracket\phi \wedge \psi\rrbracket = \llbracket\phi\rrbracket \cap \llbracket\psi\rrbracket$, from which the property follows.
- If $\sigma(\Gamma) \vdash_{\mathscr{C}} a_i$ and $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket\Gamma, A_i\rrbracket$, then $a_i \in u$. Also, $\llbracket\phi\rrbracket \supseteq f \cap g \cap u\downarrow$, where $(f, u) \in \mathscr{H}\llbracket\Gamma\rrbracket$, and $(g, u) \in \mathscr{H}\llbracket A_i\rrbracket$. $f \cap g = f \cap (a_i \to g)$, and $(a_i \to g, u) \in \mathscr{H}\llbracket \bigsqcap_{i\in I} a_i \to A_i\rrbracket$, so the result follows.
- Let $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket\Gamma, a\rrbracket$. The $a \in u$, and $\llbracket\phi\rrbracket \supseteq f \cap g$, where $f \supseteq a\uparrow \cap u\downarrow$ and $(g, u) \in \mathscr{H}\llbracket\Gamma\rrbracket$. $a \to (f \wedge g) \supseteq g$, so we have the result.
- If $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket\Gamma\rrbracket$, then since $(u\downarrow, u) \in \mathscr{H}\llbracket \bigsqcap_{i\in I} a_i \to A_i\rrbracket$, the result follows.
- This is trivial, as $A(X)$ is an unrolling of $g(X)$.

Conversely, we will show that for P a hiding free program, if $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket P\rrbracket$, we can build a proof tree for it. We will do this by structural induction on P. Note that all f in our semantics are compact, thus each corresponds to a property:

- If $P = a$, then $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket a\rrbracket$ means $a \vdash_{IL} \phi'$, so ϕ is provable by cut.
- If $P = P_1 \,\|\, P_2$, then $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket P\rrbracket$ means $\llbracket\phi\rrbracket \supseteq f \cap g$, where $(f, u) \in \mathscr{H}\llbracket P_1\rrbracket$ and $(g, u) \in \mathscr{H}\llbracket P_2\rrbracket$. Now we can get the result by induction.
- If $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket \bigsqcap_{i\in I} a_i \to A_i\rrbracket$ then if $a_i \notin u$ for all i, the $\phi = \mathtt{true}$, and this can be proved by the second rule for choice on the left. If $a_i \in u$, then $\llbracket\phi\rrbracket \supseteq a_i \to f$, where $(f, u) \in \mathscr{H}\llbracket A_i\rrbracket$. Now by applying the first rule for choice and then the cut rule, we have the result.
- If P is defined recursively, the $(\llbracket\phi\rrbracket, u) \in \mathscr{H}\llbracket P\rrbracket$ means that $(\llbracket\phi\rrbracket, u)$ is in one of the n-fold unfoldings of P. By using the rule for unfolding n times, we can use the other rules to establish the result. □

Since the singleton sets are primes in the relational powerdomain, the above theorem actually yields

$$\mathscr{H}\llbracket P\rrbracket = \{(\llbracket\phi\rrbracket, u) \mid P \Vdash_u \phi\}$$

4.5.1. Hiding

In order to accommodate hiding, we can extend the syntax of properties to include $\exists X\,.\,\phi$. We replace $\phi \vdash_{IL} \psi$ by $[\![\phi]\!] \subseteq [\![\psi]\!]$, since intuitionistic logic rules are not complete for hiding in closure operators. Now, we can write the following rules for hiding:

$$\frac{\Gamma, A[Y/X] \Vdash_v \psi \quad [\![\exists Y.\psi]\!] = [\![\phi]\!],\ \exists_Y v = u, [\![\psi]\!](u) = v,\ Y \text{ new in } \Gamma, \phi}{\Gamma, \exists X\,.\,A \quad \Vdash_u \phi}$$

$$\frac{\Gamma \Vdash_u \phi[Y/X]}{\Gamma \Vdash_u \exists X\,.\,\phi}$$

However, for this rule to be applicable we need a procedure to decide equality of closure operators. This is possible in some constraint systems, for example those which are closed under negation and forall quantification (see [24]), and then the above rules for hiding are sound.

5. 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. In this section, we sketch briefly the treatment of this notion of observation.

5.1. Operational semantics

As in the extant treatment of infinite computations in determinate cc languages [12, 24], we consider only those infinite runs of the system which are *fair* with respect to parallel composition, i.e. if in any configuration Γ at any stage in a run, there is an agent P which can make a transition, then this agent will make a transition some time during the run. Note that since enabled transitions are never disabled, the notions of weak and strong fairness [13] coincide. Furthermore, this notion of fairness *does not* impose any conditions on the choice operation.

We say o is an observation of a program P in context c if either

- $P, \Uparrow(c) \rightarrow^* \Gamma \not\rightarrow$ and $o = \exists_V.\rho(\Gamma)$, or
- There is an infinite fair sequence $P, \Uparrow(c) \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \cdots$, such that $o = \exists_V \bigcup_i \rho(\Gamma_i)$.

Definition 5.1. The must operational semantics is defined as follows:

$$\mathcal{O}_S[\![P]\!] = \{o \mid \exists o'.\|o\| = \|o'\|, o' \text{ is a observation of } P \text{ in context } \|o\|, [\![o]\!] \subseteq [\![o']\!]\}$$

The above-definition builds upclosed sets of observations. Thus, the definition allows us to identify two sets of observations which have the same minimal elements, giving us must tests [8].

5.2. Denotational semantics

5.2.1. The powerdomain

We recall the Scott topology on partial orders, referring the reader to [17] for a detailed treatment. The open sets of the Scott topology on a domain $(D, \sqsubseteq)$ are sets S of elements satisfying: (1) upwards closure: $d \in S$, $d \sqsubseteq e \Rightarrow e \in S$ and (2) limit reachability: Let E be a directed set such that $\sqcup E \in S$. Then, $E \cap S \neq \emptyset$.

A subset of S of D is Scott compact if it is compact in the Scott topology, i.e. every (Scott) open cover of S has a finite subcover.

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

- $(f,u) \in S$, $(g \cap u\downarrow) \subseteq (f \cap u\downarrow) \Rightarrow (g,u) \in S$.
- Let $S_u \stackrel{d}{=} \{(f,u) \mid (f,u) \in S\}$. Then S_u is Scott compact in the Scott topology on closure operators.

These conditions ensure that for every u, the set S_u is an element of the Smyth powerdomain [17, p. 109] of closure operators. Thus, this powerdomain inherits the basic properties of the Smyth powerdomain. The ordering relation is given by reverse subset inclusion: $S_1 \sqsubseteq S_2 \Leftrightarrow S_1 \supseteq S_2$. The greatest element is the empty set, the least element is $|\mathbf{Obs}|$, least upper bounds of directed sets are given by intersection. All bounded glb's exist, and finite greatest lower bounds are given by union. It also follows that if the union of any set of sets is compact, then their greatest lower bound is their union.

The powerdomain admits an operation $\exists_X$ defined as follows. Let S be an element of the powerdomain. If $\exists_X u = \exists_X v$, define $S_v^u = \{(f,u) \in \mathbf{Obs} \mid \exists (h,u) = (f,u).\exists (g,v) \in S, \exists_X f = \exists_X h\}$. Now $\exists_X S \stackrel{d}{=} \bigcup_u \sqcap \{S_v^u\}$. Note that as S_v is Scott compact, S_v^u is also Scott compact.[4]

5.2.2. Denotational semantics

We give the semantics of the various program combinators below:

$$\mathscr{S}[\![a]\!] \stackrel{d}{=} \{(f,u) \in \mathbf{Obs} \mid f \cap u\downarrow \subseteq a\uparrow,\ a \in u\}$$

$$\mathscr{S}[\![P \parallel Q]\!] \stackrel{d}{=} \mathscr{S}[\![P]\!] \cap \mathscr{S}[\![Q]\!]$$

$$\mathscr{S}[\![[\!]_{i\in I} a_i \to P_i]\!] \stackrel{d}{=} \{(f,u) \in \mathbf{Obs} \mid \forall i \in I. a_i \notin u\} \cup \bigcup_{i\in I} \{(f,u) \in \mathbf{Obs} \mid a_i \in u,\ \exists (f\,1', v) \in \mathscr{S}[\![P_i]\!],\ f \cap u\downarrow \subseteq a_i \to f'\}$$

$$\mathscr{S}[\![\exists X\,.P]\!] \stackrel{d}{=} \exists_X \mathscr{S}[\![P]\!]$$

[4] This follows from the general fact that if p is a continuous projection ($p(S) \leqslant S$, $p^2 = p$), then if S is Scott compact then so is $p(S)$ and $p^{-1}p(S)$ since $p(S) \subseteq p^{-1}p(S) \subseteq p(S)\uparrow$. Note also that the step $(f,u) = (h,u)$ closes the set S_v^u under equivalent bco's – this step also preserves compactness as S_v is already closed under this operation, and thus for each u has a minimal bco $(\sqcap f, u)$ (recall that the glb preserves equivalence). Now since $\exists_X$ is monotone, compactness follows.

$$\mathscr{S}[\![g(Y)]\!] \stackrel{d}{=} \mathscr{S}[\![\exists X_g.[X_g = Y \parallel g]]\!]$$

$$\mathscr{S}[\![g]\!] \stackrel{d}{=} \mu g.\mathscr{S}[\![C[g]]\!], \quad \text{where } g(X):\ C[g] \text{ is the procedure declaration}$$

Procedure calls are handled as in the case of the may semantics. All the above operations on process are monotone and continuous with respect to the ordering on sets of processes. Thus, recursion is treated in the usual way, via least fixed points.

We draw the reader's attention to the use of the bound (in a bounded closure operator) to resolve the choice in the guarded choice operator. We note that the alternative definition of parallel composition as in the case of the may semantics:

$$\mathscr{S}[\![P \parallel Q]\!] \stackrel{d}{=} \{(h,u) \in \mathbf{Obs} \mid (f,u) \in \mathscr{S}[\![P]\!],\ (g,u) \in \mathscr{S}[\![Q]\!],\ h \cap u\downarrow \subseteq f \cap g\}$$

would yield the same result. Later in the proof of the full abstraction theorem we will show that $\mathscr{S}[\![\exists X\,.P]\!] = \{(f,u) \in \mathbf{Obs} \mid \exists (g,v) \in \mathscr{S}[\![P]\!],\ \exists_X u = \exists_X v,\ \exists_X h = \exists_X g,\ (f,u) = (h,u)\}$, which is the same as the definition in the may semantics, except that since the elements of the powerdomain are upwards closed, the last condition $g(\exists_X v) = v$ is not needed.

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 may semantics, and draw the readers attention to the differences.

Example 5.2. Let P_i be a collection of processes indexed by i. Let a be a token in the constraint system. Then $\mathscr{S}[\![a \to P_i]\!] \sqsupseteq \mathscr{S}[\![\Box_{i\in I}\, a \to P_i]\!]$. 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.

Example 5.3. Let the process P_1 be defined recursively as $P_1 :: P_1$. Then the $\mathscr{S}[\![P_1]\!] = \bigcap_i \mathscr{S}[\![C^i(\mathbf{Obs})]\!]$, where $C(X) = X$. Thus $C^i(\mathbf{Obs}) = \mathbf{Obs}$, so $\mathscr{S}[\![P_1]\!] = \mathbf{Obs}$.

The following 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.

Example 5.4. Let the process P_2 be defined recursively as $P_2 :: b \parallel P_2$. Then $\mathscr{S}[\![P_1]\!] = \bigcap_i \mathscr{S}[\![C^i(\mathbf{Obs})]\!]$, where $C(X) = b \parallel X$. Thus $C(\mathbf{Obs}) = \mathscr{S}[\![b]\!]$, so $\mathscr{S}[\![P_2]\!] = \mathscr{S}[\![b]\!]$. Then $\mathscr{S}[\![P_1]\!] = \mathscr{S}[\![b]\!]$.

Example 5.5. Let P be any process. Consider the process $P_3 = b \to P \,\Box\, b \to P_1$, where P_1 is as above. Then $\mathscr{S}[\![P_3]\!] = \mathscr{S}[\![b \to P_1]\!] = |\mathbf{Obs}|$. This further clarifies the treatment of non-determinism in the must 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.

Example 5.6. The operation of adding constraints is idempotent – a key ingredient of the cc paradigm

$$\mathscr{S}[\![a]\!] = \mathscr{S}[\![a \parallel a]\!]$$

The next example shows another characteristic feature of the must style semantics – an "upward closure" property.

Example 5.7. Let $P = \mathtt{true} \to a \,[]\, \mathrm{true} \to b \to b$ and $Q = \mathtt{true} \to a$, where $b \vdash a$. Then P and Q are indistinguishable as the extra output of Q, i.e. $b^{\mathtt{true}}$ is greater than the output $a^{\mathtt{true}}$ of P, and we observe only minimal outputs.

Example 5.8. The following equational laws hold:

$$\mathscr{S}[\![a \parallel P]\!] = \mathscr{S}[\![a \parallel a \to P]\!]$$

$$\mathscr{S}[\![b \to (A \parallel B)]\!] = \mathscr{S}[\![b \to A \parallel b \to B]\!]$$

5.3. Full abstraction

5.3.1. Fair scheduling algorithms

A *fair scheduling algorithm* is an algorithm which starts with a given configuration and selects a transition to be done at each step, and the resulting run is fair. An example is a round robin scheduler.

For the simple notion of fairness that we use, the choice of fair scheduling algorithm does not matter, i.e. any fair scheduling algorithm produces all the outputs of the operational semantics.

Lemma 5.9. *The set $\mathscr{O}_S[\![P]\!]$ is the same for any fair scheduling algorithm.*

Proof. Consider a fair derivation $T - P, \Uparrow(\|o\|) \to^*$, with output o. Given a fair scheduling algorithm A, we can permute the transitions in T to produce a derivation according to A. Since $\|o\|$ is in the store at all times, the store information required to enable any transition in T is available at all times. Consider the first point when A schedules a transition different from the one that occurred in T. This transition must occur later in T as it is enabled and T is fair. Now the hypotheses of Lemma 3.8 are satisfied so we can permute T to do this transition at the current point, and by continuing this process, we can get a permutation of T which is according to the algorithm A. Thus o is an output of $A - A$ is fair so every transition of T is scheduled at some time.

The above lemma allows us to work in the context of a fixed scheduling algorithm. Consider the transitions of $P, \Uparrow(\mathtt{false})$ under a given scheduling algorithm. These can

be arranged into a tree $Tree(P)$, whose nodes are configurations, and if $\Gamma \to \Gamma'$ then Γ' is a child of Γ. Since only nondeterministic choice causes branching, this tree is finitely branching. □

Lemma 5.10. *Let o be an observation of P in context $\|o\|$. Consider all runs of the process $P, \Uparrow(\|o\|)$ that result in observations o' such that $\|o\| = \|o'\|$. These runs can be embedded in the finitely branching tree $Tree(P)$.*

Proof. The set of all such runs of the system can be embedded as a subtree in $Tree(P)$ by restricting the choice branches of $Tree(P)$. A choice branch is not present in the subtree if one of the following holds:

- it is not enabled by $\|o\|$,
- all runs in the subtree led to by the choice branch result in stores with more information than $\|o\|$.

The tree constructed above is a prefix of $Tree(P)$, i.e. if a node of $Tree(P)$ is in the subtree, so are all its ancestors. □

Corollary 5.11. *Given any set of observations O, all runs of the processes $P, \Uparrow(\|o\|)$, $o \in O$ can be arranged into a finitely branching subtree of $Tree(P)$.*

As in the previous case, we obtain full abstraction. Again we identify an observation o with the bco $(\llbracket o \rrbracket, \|o\|)$.

Theorem 5.12.

$$\mathcal{O}_S\llbracket a \rrbracket = \mathcal{S}\llbracket a \rrbracket$$

$$\mathcal{O}_S\llbracket P \parallel Q \rrbracket \stackrel{d}{=} \mathcal{O}_S\llbracket P \rrbracket \cap \mathcal{O}_S\llbracket Q \rrbracket$$

$$\mathcal{O}_S\llbracket []_{i\in I} a_i \to P_i \rrbracket \stackrel{d}{=} \{(f,u) \in \mathbf{Obs} \mid \forall i \in I. a_i \notin u\} \cup \bigcup_{i\in I} \{(f,u) \in \mathbf{Obs} \mid a_i \in u, \exists (f',u) \in \mathcal{O}_S\llbracket P_i \rrbracket, f \cap u{\downarrow} \subseteq a_i \to f'\}$$

$$\mathcal{O}_S\llbracket \exists X . P \rrbracket = \exists X \mathcal{O}_S\llbracket P \rrbracket$$

$$\mathcal{O}_s\llbracket \mu X . C[X] \rrbracket = \textit{the least fixed point of } \mathcal{O}_S\llbracket C[] \rrbracket$$

Proof.

- If $(f,u) \in \mathcal{O}_S\llbracket a \rrbracket$, then $f \cap u{\downarrow} \subseteq \llbracket a, u^u \rrbracket$. Thus $f \cap u{\downarrow} \subseteq a{\uparrow}$, and $a \in u$. Conversely, if $f \cap u{\downarrow} \subseteq a{\uparrow}$, and $a \in u$ then as a, $u^u \not\to$ and $f \cap u{\downarrow} \subseteq \llbracket a, u^u \rrbracket$, we have $(f,u) \in \mathcal{O}_S\llbracket a \rrbracket$.
- If $(f,u) \in \mathcal{O}_S\llbracket P \parallel Q \rrbracket$, then there is an observation o of $P \parallel Q$ such that $(f,u) \subseteq (\llbracket o \rrbracket, \|o\|)$. Now since $P \parallel Q$, $u^u \to P$, Q, u^u, we can create a derivation for P by picking up all transitions from P and its derivatives from the derivation for $P \parallel Q$. If the observation on this derivation is o', it follows by monotonicity and extensivity that $\|o'\| = u$ and $\llbracket o \rrbracket \subseteq \llbracket o' \rrbracket$. Thus $(f,u) \in \mathcal{O}_S\llbracket P \rrbracket$. Similarly $(f,u) \in \mathcal{O}_S\llbracket Q \rrbracket$.

Conversely, if $(f,u)\in\mathcal{O}_S[\![P]\!]$ and $(f,u)\in\mathcal{O}_S[\![Q]\!]$, suppose o_1 and o_2 are the observations of P and Q with $(f,u)\subseteq([\![o_1]\!],\|o_1\|)$ and $(f,u)\subseteq([\![o_2]\!],\|o_2\|)$. Thus by monotonicity, $o_1\wedge o_2$ is an observation of a run of $P\,\|\,Q$ obtained by fairly merging (even alternating) the runs which produced o_1 and o_2. Now since $(f,u)\subseteq([\![o_1\wedge o_2]\!],\|o_1\wedge o_2\|)$, we have $(f,u)\in\mathcal{O}_S[\![P\,\|\,Q]\!]$.

- Let $(f,u)\in\mathcal{O}_S[\![[]_{i\in I}a_i\to P_i]\!]$. Let o be a must observation of $[]_{i\in I}a_i\to P_i$, with $(f,u)\subseteq([\![o]\!],\|o\|)$. If $\forall i\in I.a_i\notin u$, then (f,u) is in the RHS. Otherwise, $[]_{i\in I}a_i\to P_i$, $u^u\to P_i^{a_i}$, u^u while making the observation o. Thus $a_i\in u$. Also, $o=a_i\to o',u^u$, where o',u^u is an observation of P_i. Thus $(a_i\uparrow\cap f,u)\subseteq([\![o']\!],u)$, so $(a_i\uparrow\cap f,u)\in\mathcal{O}_S[\![P_i]\!]$. Since $f\subseteq a_i\to(a_i\uparrow\cap f)$ we have the result.
  The converse follows by reversing the above arguments.
- We first show the following lemma. □

Lemma 5.13. $\exists_X\mathcal{O}_S[\![P]\!]=\{(f,u)\in\mathbf{Obs}\mid\exists(h,u)=(f,u).\exists(g,v)\in\mathcal{O}_S[\![P]\!].\ \exists_X u=\exists_X v,\exists_X h=\exists_X g\}$.

Proof. Let $S=\mathcal{O}_S[\![P]\!]$. We need to show that $(\exists_X S)_u=\{(f,u)\mid\exists(g,v)\in\mathcal{O}_S[\![P]\!],\ \exists_X u=\exists_X v,\ \exists_X h=\exists_X g,\ (h,u)=(f,u)\}$. Since the right-hand side is just $\bigcup\{S_v^u\mid\exists_X v=\exists_X u\}$, we need to show that $\bigcup\{S_v^u\mid\exists_X v=\exists_X u\}=\sqcap\{S_v^u\mid\exists_X v=\exists_X u\}$, that is we must show that $\bigcup\{S_v^u\mid\exists_X v=\exists_X u\}$ is Scott compact. From the corollary to Lemma 5.10 for P, all the derivations leading to any $(g,v)\in\mathcal{O}_S[\![P]\!]$, $\exists_X v=\exists_X u$ are embeddable in the finitely branching tree constructed there. To each node of the embedded subtree we associate a bco (f,u) where f is the contexted store at the node after hiding X and any new variables. Thus all elements of all S_v^u's are embedded in this tree either as leaf nodes or as limits of infinite paths of nodes (or are included by upwards closure). A Scott open set which contains a node will contain the entire subtree below this node, and any open set that contains the limit of a path must contain a node on that path. The Scott compactness of $\bigcup\{S_v^u\mid\exists_X v=\exists_X u\}$ follows from Konig's lemma on the finite branching tree.

The proof now follows the proof in the may semantics. Let $(f,u)\in\mathcal{O}_S[\![\exists X.P]\!]$ such that o is an observation of $\exists X.P$ satisfying $(f,u)\subseteq([\![o]\!],\|o\|)$. This means that there is a derivation $\exists X.P,\ u^u\to P[Y/X],\ u^u\to\cdots$ such that

- Y is new in u and P,
- $o=\exists_X o'$, u^u, where o' is the observation of the derivation $P,\Uparrow(()\exists_X u)\to\cdots$ corresponding to the above derivation $P[Y/X],\ u^u\to\cdots$.
  Since $\exists_X u\subseteq\|o'\|$, we have $\exists_X o=\exists_X o'\sqcup\exists_X u^u=\exists_X o'$.

Conversely, let $(g,v)\in\mathcal{O}_S[\![P]\!]$. Note that given any $(g,v)\in\mathcal{O}_S[\![P]\!]$, we can choose $g'=(g\,\|\,\exists_X v\to v)$ – then $(g',v)\in\mathcal{O}_S[\![P]\!]$ by upwards closure, $g'(\exists_X v)=v$, and $\exists X.g=\exists X.g'$. Let (f,u) be such that $\exists X.g=\exists X.h,\exists_X u=\exists_X v,(h,u)=(f,u)$. We will show that $(f,u)\in\mathcal{O}_S[\![\exists X.P]\!]$.

Since $(g,v)\in\mathcal{O}_S[\![P]\!]$, we deduce that there is a derivation starting at $P,v^v\to\cdots\to\cdots$, with output o such that $g\cap v\downarrow\subseteq[\![o]\!]$. Since, $g(\exists_X v)=v$, we have $[\![o]\!](\exists_X v)=v$; using

Lemma 3.12, we deduce that P, $(\exists_X v)^{\exists_X v} \to \cdots \to \cdots$. This sequence of reductions can be mimicked by $\exists X\,.\,P, u$, and the result follows.

- If $P = \mu X\,.\,C[X]$, then we want to show that $\mathcal{O}_S[\![P]\!]$ is the least fixed point of $\mathcal{O}_S[\![C[]]\!]$, which is an operator from processes to processes. By the above proofs, it follows that $C[]$ is a continuous operator, thus it has a least fixed point. Also by the rules given above, we can show that

$$\mathcal{O}_S[\![C[]]\!](\mathcal{O}_S[\![Q]\!]) = \mathcal{O}_S[\![C[Q]]\!]$$

for any program Q. It follows that the least fixed point of $\mathcal{O}_S[\![C[]]\!]$ is $\bigcap_i \mathcal{O}_S[\![C^i[\mathtt{true}]]\!]$, where `true` is the least element of the powderdomain, **Obs**.

Consider the derivations of P, u^u. These can be arranged into a finitely branching tree T using Lemma 5.10. To each node of the embedded subtree we associate a bco (f, u) where f is contexted store at the node after hiding any new variables. Thus all elements of the from $(h, u) \in \mathcal{O}_S[\![P]\!]_u$ are in this tree either as leaf nodes or as limits of infinite paths of nodes (or are included by upwards closure). Furthermore, the derivation tree of each $C^i[\mathtt{true}], u^u$ can be viewed as a subtree of T. In fact, if $i \leqslant j$, the tree for $C^i[\mathtt{true}], u^u$ is a prefix of the one for $C^j[\mathtt{true}], u^u$, and every node in T arises from some node in $C^i[\mathtt{true}]$ for some i.

If $(f, u) \in \bigcap_i \mathcal{O}_S[\![C^i[\mathtt{true}]]\!]$ it is in each $\mathcal{O}_S[\![C^i[\mathtt{true}]]\!]$. Thus, there are derivations of o_i from each of $C^i[\mathtt{true}], u^u$, such that $f \subseteq [\![o_i]\!]$. We now construct a run of P, u^u as follows. We begin at the root of the tree. Any non-choice transition is performed as in the tree. At a choice node, we choose any transition that is chosen in infinitely many i's in derivations of o_i from each of $C^i[\mathtt{true}], u^u$, such that $f \cap u\downarrow \subseteq [\![o_i]\!]$. By Konig's lemma, either T is finite or the above procedure constructs an infinite path in the tree. In either case, we build a derivation from P, u^u such that $f \subseteq [\![\sigma(P_k)]\!]$ at any finite stage $P, u^u \to^* P_k$.

Conversely, if $(f, u) \in \mathcal{O}_S[\![P]\!]$, we have a derivation of o such that $f \cap u\downarrow \subseteq [\![o]\!]$. By repeatedly applying the unfolding rule, it is clear that it is in each of the $\mathcal{O}_S[\![C^i[\mathtt{true}]]\!]$, since for the corresponding observation o_i, $[\![o]\!] \subseteq [\![o_i]\!]$. □

The following theorem follows by structural induction.

Theorem 5.14. *If P, Q are two indeterminate programs, then*

$$\mathcal{S}[\![P]\!] = \mathcal{S}[\![Q]\!] \Leftrightarrow \mathcal{O}_S[\![P]\!] = \mathcal{O}_S]\!]Q]\!]$$

5.4. Must semantics in logical form

Intuitively, the must semantics consists of properties that are satisfied by all runs of the program. As before, we consider hiding free programs only.

The syntax of properties is derived from the following grammar:

$$\phi ::= u \mid a \to \phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

The following deduction rules establish when this is true. Once again Γ stands for a multiset of agents, and $\sigma(\Gamma)$ for the tell tokens in it:

$$\frac{\Gamma \Vdash_u \phi}{\Gamma, P \Vdash_u \phi} \quad \frac{\Gamma, A, B, \Delta \Vdash_u \phi}{\Gamma, B, A, \Delta \Vdash_u \phi} \quad \frac{\overline{\sigma(\Gamma)} \supseteq v}{\Gamma \Vdash_u v}$$

$$\frac{\Gamma, P_1, P_2 \Vdash_u \phi}{\Gamma, (P_1, P_2) \Vdash_u \phi} \quad \frac{\Gamma \Vdash_u \phi_1 \quad \Gamma \Vdash_u \phi_2}{\Gamma \Vdash_u \phi_1 \wedge \phi_2}$$

$$\frac{\sigma(\Gamma) \Vdash_{\mathscr{C}} a \quad \Gamma, A \Vdash_u \phi}{\Gamma, a \to A \Vdash_u \phi} \quad \frac{av \quad \Gamma, a \Vdash_u \phi}{\Gamma \Vdash_u a \to \phi}$$

$$\frac{J = \{i \in I \mid a_i \in u\} \neq \emptyset \qquad \forall i \in J. \Gamma, a_i \to P_i \Vdash_u \phi}{\Gamma, []_{i \in I} a_i \to P_i \Vdash_u \phi}$$

$$\frac{\Gamma, A(X) \Vdash_u \phi(g(X) :: A(X))}{\Gamma, g(X) \Vdash_u \phi} \quad \frac{\Gamma \Vdash_u \phi_1}{\Gamma \Vdash_u \phi_1 \vee \phi_2}$$

We draw the reader's attention to the rule for guarded choice. The rule ensures that *all* enabled branches of the guarded choice satisfy the property, a characteristic feature of must testing style semantics. Note that the rules given above are conservative over logical entailment, thus if $\phi \Vdash \psi$ follows from the constraint system (with standard intuitionist logic rules), then if $\Gamma \Vdash_u \phi$, then $\Gamma \Vdash_u \psi$. In particular, the rules are conservative over the logical entailment of the underlying constraint system.

The logical semantics of a program can now be defined. Suppose $P \Vdash_u \phi$. Embed ϕ in the powderdomain as $E_S(\phi)$ as follows.

Definition 5.15. $E_S(\phi)$ is defined inductively as follows:

$$E_S(c) = \{(f, u) \in \mathbf{Obs} \mid a \in u, f \subseteq \bar{a}\uparrow, c \subseteq u\}$$

$$E_S(a \to \phi) = \{(f, u) \in \mathbf{Obs} \mid a \notin u\}$$
$$\cup \{(f, u) \in \mathbf{Obs} \mid a \in u, \exists (f' v) \in \mathscr{S}[\![P_i]\!], f \cap u\downarrow \subseteq a \to f'\}$$

$$E_S(\phi_1 \wedge \phi_2) = E_S(\phi_1) \cap E_S(\phi_2)$$

$$E_S(\phi_1 \vee \phi_2) = E_S(\phi_1) \cup E_S(\phi_2)$$

Let $\mathscr{E}_S(\phi, u)$ be the subset of $E_S(\phi)$ containing those bco's with second component u (note that this set may be empty!). Then the logical semantics is given as the intersections of the embedded sets – $\mathscr{L}_S[\![P]\!] = \bigcup_u \bigcap_\phi \{\mathscr{E}_S(\phi, u) \mid P \Vdash_u \phi\}$. And we get the theorem:

Theorem 5.16. *For any hiding-free program P, $\mathscr{L}_S[\![P]\!] = \mathscr{S}[\![P]\!]$.*

Proof. The proof exploits the algebraicity of the powderdomain. Note that compact elements of the powderdomain can be viewed as finite disjunctions. We prove that the logical semantics corresponding to a compact element less than the denotation of

the program can be proved using the logical system. If $P \Vdash_u \phi$, then $\mathscr{S}[\![P]\!]_u \subseteq \mathscr{E}_S(\phi, u)$. This is proved by induction on the *finite* proof tree for $P \Vdash_u \phi$ – by showing that if the antecedent of each proof rule satisfies the inclusion, then so does the consequent. This is a straightforward application of the definition of processes. This means $\mathscr{S}[\![P]\!]_u \in \bigcap_\phi \mathscr{E}_S(\phi, u)$. Thus $\mathscr{L}_S[\![P]\!] \supseteq \mathscr{S}[\![P]\!]$.

For the converse we do a structural induction on programs, showing that for every u, $\mathscr{L}_S[\![P]\!]_u \subseteq \mathscr{S}[\![P]\!]_u$. a is provable from a, so $\mathscr{L}_S[\![P]\!]_u \subseteq \mathscr{E}_S(a, u)$. $\mathscr{E}_S(a, u)$ is empty if $a \notin u$, otherwise is equal to $\mathscr{S}[\![a]\!]$.

If $P \Vdash_u \phi$, then by weakening, $P, Q \Vdash_u \phi$. Thus $\mathscr{L}_S[\![P \parallel Q]\!]_u \subseteq \mathscr{L}_S[\![P]\!]_u$, and similarly for Q. So by induction $\mathscr{L}_S[\![P \parallel Q]\!]_u \subseteq \mathscr{S}[\![P]\!]_u \cap \mathscr{S}[\![P]\!]_u = \mathscr{S}[\![P \parallel Q]\!]_u$.

To prove the case $P = []_{i \in I} a_i \to P_i$, we first show the result for $P = a \to Q$. If $Q \Vdash_u \phi$, then $P \Vdash_u a \to \phi$. Then by using the induction hypothesis on Q we see that $\mathscr{L}_S[\![P]\!]_u \subseteq \mathscr{S}[\![P]\!]_u$.

Now if $P = []_{i \in I} a_i \to P_i$, then let ϕ_i be provable from $a_i \to P_i$, for each $i \in J = \{i \in I \mid a_i \in u\}$. Then $\bigvee_{i \in J} \phi_i$ is also provable from $a_i \to P_i$, thus it is provable from P. Now by the definition of $\mathscr{S}[\![P]\!]$, we have the result.

If $P = \mu X.\, C[X]$, we know by the induction hypothesis that $\mathscr{L}_S[\![C^n(\texttt{true})]\!] \subseteq \mathscr{S}[\![C^n(\texttt{true})]\!]$. Now any property of $C^n(\texttt{true})$ can also be proved by P by n-fold application of the unfolding rule. Thus $\mathscr{L}_S[\![P]\!] \subseteq \bigcap_n \mathscr{L}_S[\![C^n(\texttt{true})]\!] \subseteq \bigcap_n \mathscr{S}[\![C^n(\texttt{true})]\!] = \mathscr{S}[\![P]\!]$. □

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

[1] S. Abramsky, Domain theory in logical form, Ann. Pure Appl. Logic 51 (1991) 1–77.
[2] L. Aceto, M. Hennessy, Towards action-refinement in process algebras, Proc. 4th Annu. Symp. on Logic in Computer Science, IEEE Computer Society Press, Silverspring, MD, 1989, pp. 138–145.
[3] G. Boudol, I. Castellani, M.C. Hennessy, A. Kiehn, A theory of processes with localities, Proc. Internat. Conf. on Concurrency Theory, Lecture Notes in Computer Science, vol. 630, 1992, pp. 108–122.
[4] F.S. de Boer, M. Gabrielli, E. Marchiori, C. Palamidessi, Proving concurrent programs correct, Proc. 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1994, pp. 98–108.
[5] F.S. de Boer, C. Palamidessi, A fully abstract model for concurrent constraint programming, Proc. TAPSOFT/CAAP, Lecture Notes in Computer Science, vol. 493, 1991, pp. 296–319.
[6] F.S. de Boer, C. Palamidessi, E. Best, Concurrent constraint programming with information removal, Proc. Concurrent Constraint Programming Workshop, Venice, 1995, pp. 1–13.
[7] J. deKleer, An assumption based TMS, Artificial Intelligence 28 (1986) 127–162.
[8] R. de Nicola, M.C.B. Hennessy, Testing equivalences for processes, Theoret. Comput. Sci. 34 (1984) 83–133.

[9] M. Fromherz, V. Saraswat, Model-based computing: Using concurrent constraint programming for modeling and model compilation, Principles and Practices of Constraint Programming, Lecture Notes in Computer Science, vol. 976, Springer, Berlin, 1995, pp. 629–635.
[10] R. Gorrieri, Refinement atomicity, and transactions for process description languages, Ph.D. Thesis, University of Pisa, 1991.
[11] P. Van Hentenryck, V.A. Saraswat, Y. Deville, Constraint processing in cc(fd), Tech. Report, Computer Science Department, Brown University, 1992.
[12] R. Jagadeesan, P. Panangaden, K. Pingali, A fully-abstract semantics for a functional language with logic variables, ACM Trans. Programming Languages Systems 13(4) (1991) 577–625; Proc. 4th IEEE Symp. on Logic in Computer Science, June 1989 (preliminary version).
[13] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer, Berlin, 1991, 427 pp.
[14] U. Montanari, F. Rossi, True concurrency semantics for concurrent constraint programming, in: V. Saraswat, K. Ueda (Eds.), Proc. 1991 Internat. Logic Programming Symp., 1991.
[15] U. Montanari, F. Rossi, A concurrent semantics for concurrent constraint programs via contextual nets, Principles and Practices of Constraint Programming, 1995, pp. 3–27.
[16] G.D. Plotkin, A powerdomain construction, SIAM J. Comput. 5(3) (1976) 452–487.
[17] G.D. Plotkin, Domains. Available from `http://hypatia.dcs.qmw.ac.uk/sites/other/domain.notes.other/`, 1983.
[18] V.R. Pratt, Modeling concurrency with partial orders, Int. J. Parallel Programming 15(1) (1986) 33–71.
[19] V.A. Saraswat, The category of constraint systems is Cartesian-closed, Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz, 1992.
[20] V.A. Saraswat, Concurrent constraint programming, Doctoral Dissertation Award and Logic Programming Series, MIT Press, Cambridge, MA, 1993.
[21] V.A. Saraswat, R. Jagadeesan, V. Gupta, Programming in timed concurrent constraint languages, in: B. Mayoh, E. Tougu, J. Penjam (Eds.), Constraint Programming, NATO Advanced Science Institute Series F: Computer and System Sciences, Vol. 131, Springer, Berlin, 1994, pp. 367–413.
[22] M.B. Smyth, Powerdomains, J. Comput. System Sci. 16 (1978) 23–36.
[23] V.A. Saraswat, M. Rinard, Concurrent constraint programming, Proc. 17th ACM Symp. on Principles of Programming Languages, San Fransisco, January 1990.
[24] V.A. Saraswat, M. Rinard, P. Panangaden, Semantic foundations of concurrent constraint programming, Proc. 18th ACM Symp. on Principles of Programming Languages, Orlando, January 1991, pp. 333–352.
[25] R. van Glabbeek, F. Vaandrager, Petri net models for algebraic theories of concurrency, Proc. of PARLE, Lecture Notes in Computer Science, vol. 259, 1987, pp. 224–242.
[26] W. Vogler, Modular Construction and Partial Order Semantics of Petri Nets, Lecture Notes in Computer Science, vol. 625, Springer, Berlin, 1992, 252 pp.
[27] G. Winskel, Event structures, in: Petri Nets: Applications and Relationships to Other Models of Concurrency, Lecture Notes in Computer Science, vol. 255, 1987, pp. 325–392.