Prefixing distributes through parallel composition.

$$
\begin{array}{ll}
(L8) & c \wedge d = (c \sqcup d) \\
(L9) & c \rightarrow (A \wedge B) = (c \rightarrow A) \wedge (c \rightarrow B) \\
(L10) & (a \rightarrow b) \wedge (c \rightarrow d) = (a \rightarrow b) \\
 & if\ c \geq a, b \geq d \\
(L11) & (a \rightarrow b) \wedge (c \rightarrow d) = (a \rightarrow b) \wedge (c \sqcup b \rightarrow d) \\
 & if\ c \geq a \\
(L12) & (a \rightarrow b) \wedge (c \rightarrow d) = (a \rightarrow b) \wedge (c \rightarrow d \sqcup b) \\
 & if\ d \geq a
\end{array}
$$

The transition rule for $A \wedge B$ reflects the fact that A and B never communicate synchronously in $A \wedge B$. Instead, all communication takes place asynchronously with information added by one agent stored in the shared constraint for the other agent to use.

$$
\frac{A \xrightarrow{(c,d)} A'}{\begin{array}{l} A \wedge B \xrightarrow{(c,d)} A' \wedge B \\ B \wedge A \xrightarrow{(c,d)} B \wedge A' \end{array}} \tag{6}
$$

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

$$\exists X f = \{c \in |D|_0 \mid \exists d \in f . \exists_X c = \exists_X d\}$$

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

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

$$
\frac{A \xrightarrow{(\exists_X c, d)} B}{\exists X A \xrightarrow{(c, c \sqcup \exists_X d)} \exists X(d, B)} \tag{7}
$$

This axiom reflects the fact that all information about X in c is hidden from $\exists X A$, and all information about X that A produces is hidden from the

environment. Note that B may need the produced information about X to progress; this information is stored with B in the constraint d.

The axiom for agents with an internal store is straightforward. The information from the external store is combined with information in the internal store, and any new constraint generated in the transition is retained in the internal store:

$$\frac{A \xrightarrow{(d \sqcup \exists_X c, d')} B}{\exists X(d, A) \xrightarrow{(c, c \sqcup \exists_X d')} \exists X(d', B)} \tag{8}$$

In order to canonicalize agents with this operator, we need the following law:

$$(Ex1) \quad \exists X c = \exists_X c$$

In order to get a complete equational axiomatization for finite agents containing subagents of the form $\exists XB$, we need the constraint system to be expressive enough. Specifically, we require:

(C1) For all $c \in |D|_0$ and $X \in Var$, there exists $d \in |D|_0$ (written $\forall_X c$) such that for all $d' \in |D|_0$, $d' \geq d$ iff $\exists_X d' \geq c$.

(C2) For all $c, c' \in |D|_0$ and $X \in Var$, there exists a $d \in |D|_0$ (written $\Rightarrow_X (c, c')$) such that for all $d' \in |D|$, $c \cup \exists_X d' \geq c'$ iff $\exists_X c \cup \exists_X d' \geq d$.

Now we can state the remaining laws needed to obtain a complete axiomatization.

$$\begin{array}{ll} (Ex2) & \exists X(c \rightarrow A) = \forall_X(c) \rightarrow \exists X.A \\ (Ex3) & \exists X \wedge_{i \in I} c_i \rightarrow d_i = \\ & \wedge_{i \in I} \exists X(c_i \rightarrow (d_i \wedge_{j \in I, j \neq i} c_j \rightarrow d_j)) \\ (Ex4) & \exists X(c \wedge_{i \in I} c_i \rightarrow d_i) = \\ & \exists_X c \wedge \exists X \wedge_{i \in I} \Rightarrow_X (c, c_i) \rightarrow d_i \end{array}$$

Recursion. Recursion is handled in the usual way, by taking limits of the denotations of all syntactic approximants, since the underlying domain is a cpo and all the combinators are continuous in their process arguments.

Operationally, procedure calls are handled by looking up the procedure in the environment ρ. The corresponding axiom is:

$$\rho \vdash p(X) \xrightarrow{(d,d)} \exists \alpha(d_{\alpha X}, \rho(p)) \tag{9}$$

Example 3.1 (Append) To illustrate these combinators, consider the `append` procedure in the determinate `cc` language, using the Kahn constraint system:

```
append(In1, In2, Out) ::
    In1 = A → Out = In2
    ∧ c(In1) → ∃X (Out = a(f(In1), X) ∧ append(r(In1), In2, X)).
```

This procedure waits until the environment either equates X to Λ or puts at least one data item onto the communication channel X. It then executes the appropriate branch of the body. Note that because the ask conditions in the two branches are mutually exclusive, no call will ever execute the entire body of the procedure. This procedure therefore uses the $\wedge$ operator (which ostensibly represents parallel execution) as a determinate choice operator. This is a common idiom in determinate concurrent constraint programs. □

Completeness of axiomatization. Completeness of axiomatization is proven via the following "normal form".

Definition 3.2 An agent A is in normal form iff $A = \texttt{true}$ or $A = \wedge_{i \in I} c_i \rightarrow d_i$ and A satisfies the following properties:

$$\begin{array}{ll} (P1) & c_i < d_i \\ (P2) & i \neq j \text{ implies } c_i \neq c_j \\ (P3) & c_i < c_j \text{ implies } d_i < c_j \\ (P4) & c_i \leq d_j \text{ implies } d_i \leq d_j \end{array}$$

□

Lemma 3.1 *Any agent A containing no constructs of the form $\exists XB$ can be converted to normal form using equations $(L1) - (L12)$.*

Lemma 3.2 *For any agent $A = \wedge_{i \in I} c_i \rightarrow d_i$ in normal form, $\mathcal{P}(\epsilon.A)(c_i) = d_i$.*

We use this lemma when proving the following completeness theorem:

Theorem 3.3 *$\mathcal{P}(\epsilon.A) = \mathcal{P}(\epsilon.B)$ iff A and B have the same normal form.*

Thus the laws $(L1) \ldots (L12)$ are both sound and complete for finite agents built using tells, asks and parallel composition.

In addition, we also have:

Theorem 3.4 *Laws $(L1)$ - $(L12)$ and $(Ex1)$-$(Ex4)$ are both sound and complete for all finite agents.*

Operational semantics. In order to extract an environment from $D.A$ in which to run A, we define:

$$\mathcal{R}(\epsilon)\rho = \rho$$
$$\mathcal{R}(p(X) :: A.D)\rho = \mathcal{R}(D)\rho[p \mapsto (\exists X d_{\alpha X} \wedge A)]$$

A computation in this transition system is a sequence of transitions in which the environment is constrained to produce nothing. Hence the final constraint of each transition should match the initial constraint of the succeding transition. The following definition formalizes the notion of a computation starting from a finite constraint c:

Definition 3.3 A c-sequence s for a program $D.A$ is a possibly infinite sequence $\langle c_i, A_i \rangle_i$ of pairs of agents and stores such that $c_0 = c$ and $A_0 = A$ and for all i, $\mathcal{R}(D)\rho_0 \vdash A_i \stackrel{(c_i, c_{i+1})}{\longrightarrow} A_{i+1}$. Here ρ_0 is the partial map from procedure names to (syntactic) agents whose domain is empty. Such a transition sequence is said to be *terminal* if it is finite, of length $n \geq 1$ and A_{n-1} is *stuck* in c_{n-1} (that is, there is no constraint d and agent B such that $\mathcal{R}(D)\rho_0 \vdash A_{n-1} \stackrel{(c_{n-1}, d)}{\longrightarrow} B$). In such a case, c_{n-1} is also called the *final* store. □

One can prove a number of operational results in a fairly straightforward way.

Lemma 3.5 (Operational monotonicity.) *If $A_1 \stackrel{(c,d)}{\longrightarrow} A_2$ is a possible transition and $e \geq c$ then $A_1 \stackrel{(e, d \sqcup e)}{\longrightarrow} A_2$.*

This is essentially an operational *monotonicity* property.

Definition 3.4 Suppose that an agent A in a store c has two transitions enabled, i.e. it could do either one of $A \stackrel{(c,c_1)}{\longrightarrow} A_1$ and $A \stackrel{(c,c_2)}{\longrightarrow} A_2$. We say that these transitions commute if $A_1 \stackrel{(c_1, c_1 \sqcup c_2)}{\longrightarrow} A_3$ and $A_2 \stackrel{(c_2, c_1 \sqcup c_2)}{\longrightarrow} A_3$ are both possible. □

The following lemma is almost immediate and characterizes a key property of determinate agents.

Lemma 3.6 *If an agent has more than one transition possible in a given store they will commute.*

The following theorem can be proved by appealing to commutativity.

Theorem 3.7 (Confluence) *For any constraint c and determinate program $D.A$, if $D.A$ has a terminal c-sequence with final store d, then $D.A$ has no infinite c-sequence. Further, all terminal c-sequences have the same final store.*

This theorem allows us to define an observation function on programs mapping $|D|_0$ to $|D|_0$ by: $\mathcal{O}(P)(c) = d$ if P has a terminal c-sequence with final store d, and $\mathcal{O}(P)(c) = \texttt{false}$ otherwise.

Theorem 3.8 *The function $\mathcal{O}(P)$ is a closure operator.*

The only nontrivial part of this proof is showing idempotence; it is done by induction on the length of reduction sequences and use of Lemma 3.5.

We can now connect the operational semantics with the denotational semantics.

Theorem 3.9 (Strong adequacy) $\mathcal{O}(P) = \mathcal{P}(P)$

Therefore, two programs P and Q are observationally equal (that is, $\mathcal{O}(P) = \mathcal{O}(Q)$) iff their denotations are equal (that is, $\mathcal{P}(P) = \mathcal{P}(Q)$). Thus the denotations of programs contain enough information to distinguish programs that are operationally different.

Proof sketch: One can show that a single reduction step preserves the denotational semantics. Then we show that the sets of fixed points of the two closure operators are the same. In order to do this we use a structural induction and a fixed-point induction for the recursive case. The proofs are not trivial but they are not particularly novel either. The full paper will contain a more thorough discussion.

It remains to show that the denotations of two programs are identified if, from the viewpoint of the operational semantics, they behave identically in all *contexts*.

Definition 3.5 A context $\mathcal{C}[\bullet]$ is a program $D.A[\bullet]$ whose agent A contains a "placeholder" (denoted by $\bullet$). We put a program $D'.A'$ into this context by taking the union of the definitions (renaming procedures where necessary to avoid name clashes) and replacing the placeholder $\bullet$ in A with A', yielding $\mathcal{C}[D'.A'] = D \cup D'.A[A']$. □

Theorem 3.10 (Full abstraction) $\mathcal{P}(P) = \mathcal{P}(Q)$ *iff for all contexts* $\mathcal{C}[\bullet]$, $Obs(\mathcal{C}[P]) = Obs(\mathcal{C}[Q])$.

The theorem is easy to prove given that we have strong adequacy and a compositional definition of the denotational semantics.

3.2 Alternate semantic treatments

The first semantics is based on the notion that it is appropriate to confuse the process that takes some input c to `false` and halts, with the process that diverges on input c. However, several other coherent alternative notions for handling divergence can be modelled with minor variations on the above theme. In this section we show briefly how to generate a model which distinguishes between `false` and **div**, and also how to generate a model which associates with each input the limit of fair execution sequences of the program on that input. In each case we sketch the major idea and leave a full development as an exercise for the reader.

Distinguishing div from `false`. Suppose for each input to a process we observe whether or not the process diverges, and if it does not, we observe the resultant store. Thus, the denotation f of an agent A will be a *partial* function from $|D|_0$ to $|D|_0$. What sort of function? Observe that if a determinate `cc` process engages in an infinite execution sequence in c, then it must also engage in an infinite execution sequence in a store $d \geq c$. Therefore the domain of f will be downward-closed. However, as before, on this domain f will be a closure operator. This motivates the definition:

Definition 3.6 A *partial closure operator* on a lattice E is a closure operator on a downward-closed subset of E. □

As before, the range of a partial closure operator contains enough information to recover the function. In particular, the domain of the function is just the downward closure of the range of the function. Thus, the added generality arises merely from the fact that `false` is not required to be in the range of (and hence a fixed point of) a partial closure operator!

Note that the (range of the) partial closure operator corresponding to **div**, the program that diverges on every input, is just $\emptyset$, since the domain of the function is the empty set. On the other hand, the (range of the) partial closure operator corresponding to `false` is {`false`}. Thus this semantics distinguishes between these two programs.

As before, partial closure operators can be partially ordered by the divergence ordering:

$$f \leq g \iff f \subseteq g \subseteq f \cup df$$

where df, the set of inputs on which f is undefined is just the complement in $|D|_0$ of the domain of f (i.e., $|D|_0 \setminus \downarrow f$).

Rather surprisingly, the definition of the combinators remains *unchanged*, even though the "meaning" (operational interpretation) of the denotation has changed:

$$\begin{aligned}
c\star &= \{d \in |D|_0 \mid d \geq c\} \\
c \rightarrow A &= \{d \in |D|_0 \mid d \geq c \Rightarrow d \in A\} \\
A_1 \wedge A_2 &= \{d \in |D|_0 \mid d \in A_1 \wedge d \in A_2\} \\
\exists X A &= \{d \in |D|_0 \mid \exists c \in A . \exists_X c = \exists_X d\}
\end{aligned}$$

Each of these definitions yields a partial closure operator when its process arguments are partial closure operators. Each defines a function that is monotone and continuous in its process arguments.

Connections with the operational semantics can be established in a manner analogous to the connections established above.

A semantics based on observing limits. The above semantics treats a divergent computation as catastrophic—it is treated as the computation that causes the store to become inconsistent. As discussed earlier, it is possible to

develop a different semantics, one in which limits of fair execution sequences are observed. For example, such a semantics would associate the cc/Kahn process:

```
ones(X) :: ∃Y X = a(1.A, Y) ∧ ones(Y).
```

with the closure operator that maps `true` to the (limit) constraint that forces X to be the infinite sequence of 1s, whereas the previous semantics would associate this program with the partial closure operator that diverges in `true`.

First we need to define the notion of fair execution sequence. At any stage of the computation there may be several enabled transitions, each of which reduces one of the agent's subagents. Note that if a subagent can be reduced at a given stage of the computation, it can be reduced at every successive stage of the computation until a transition is taken that actually carries out the reduction. We say that a c-sequence s is *fair* if it eventually reduces every subagent that can be reduced at some stage of s. This is a common notion that one needs in defining the operational semantics of concurrent systems.

In such a semantics, the denotation of a process associates with each input the limit of the sequence of store on any fair execution sequence of the process. Hence, the denotation is taken to be an operator over $|D|$ (instead of over $|D|_0$). As above, the denotation must be a closure operator—but in addition, it seems reasonable to demand that no process can decide to produce some output after it has examined an infinite amount of input. That is, we demand that f be *continuous*: for every directed $S \subseteq D$:

$$f(\sqcup S) = \sqcup f(S) \tag{10}$$

In terms of fixed points, it is not hard to see that if S is the set of fixed points of a closure operator f, then f is continuous iff S is closed under lubs of directed subsets.

The partial order on processes is now the extensional one: $f \sqsubseteq g$ iff $f \supseteq g$. The bottom element in the partial order is Id (that is, $|D|$) (thus limits of chains are given by interesection) and the top element is $\{$`false`$\}$, the operator which maps every element to `false`.

Even more surprisingly, the definition of combinators remains unchanged from the previous section, modulo the fact that fixed points must now be taken from $|D|$ instead of just $|D|_0$:

$$\begin{aligned}
&c\star = \{d \in |D| \mid d \geq c\} \\
&c \rightarrow A = \{d \in |D| \mid d \geq c \Rightarrow d \in A\} \\
&A_1 \wedge A_2 = \{d \in |D| \mid d \in A_1 \wedge d \in A_2\} \\
&\exists X A = \{d \in |D| \mid \exists c \in A . \exists_X c = \exists_X d\}
\end{aligned}$$

Each of these combinators is seen to be well-defined (they yield continuous closure operators when their process arguments are continuous closure operators), and monotone and continuous in their process arguments.

The following result follows from the commutativity properties of transitions.

Theorem 3.11 *If s_1 and s_2 are both fair c-sequences for A, then $\sqcup Cons(s_1) = \sqcup Cons(s_2)$, where $Cons(s)$ yields the set of constraints from s.*

This theorem allows us to define an observation function on programs mapping $|D|_0$ to $|D|$ by:

$$Obs(P)(c) = \sqcup Cons(s)$$

for s any fair c-sequence for P.

Relationship with the denotational semantics. This discussion is quite brief as it is quite similar to the previous discussion. The new issues one must to deal with are that transition sequences have to be fair and the semantic domain has an entirely different order. Also, the operational semantic function is defined on the entire domain generated by the constraint system rather than just the finite elements.

The relevant theorems are as follows.

Theorem 3.12 *$\mathcal{O}(P)$, the continuous extension of $Obs(P)$ is a closure operator on $|D|$.*

Theorem 3.13 *$\mathcal{O}(P) = \mathcal{P}(P)$.*

Theorem 3.14 (Full abstraction) *$\mathcal{P}(P) = \mathcal{P}(Q)$ iff for all contexts $\mathcal{C}[\bullet]$, $Obs(\mathcal{C}[P]) = Obs(\mathcal{C}[Q])$.*

4 The nondeterminate language

Let us now consider the determinate cc language in the previous section, together with (bounded) nondeterminate choice. Syntactically, admit as an agent expressions of the form

$$c_1 \rightarrow A_1 \,\square\, c_2 \rightarrow A_2 \,\square\, \ldots \,\square\, c_n \rightarrow A_n$$

for finite constraints c_i and agents A_i, $n \geq 1$. Intuitively, in any store d, such an agent can in one uninterruptible step reduce to A_i without affecting the store provided that the ith branch is "open", that is, $d \geq c_i$. If no branch is open, the agent remains stuck, and if more than one branch is open, then any one can be chosen. Thus the axiom for dependent choice satisfied by the $\longrightarrow$ relation is:

$$\square_{j \in J}(c_j \rightarrow A_j) \xrightarrow{(d,d)} A_j \qquad \text{if } d \geq c_j, \text{ for some } j \in J \tag{11}$$

With this construct admitted into the language, the denotation of an agent can no longer be a function from $|D|_0$ to $|D|_0$. Neither can it be just a relation in $|D|_0 \times |D|_0$, since parallel composition will not be definable. Instead we model a process as a set of *failures*, which record the interactions that a process engages in with the environment before reaching a state ("resting point") in which it cannot progress without intervention by the environment. This simple idea turns out to be adequate to give us a denotational semantics which is fully abstract with respect to a notion of observation that includes observation of divergence and the final (quiescent) stores of an execution sequence.

The rest of this section is devoted to giving an exposition of this model. Because of the nature of constraint-based communication, it turns out to be very convenient to model failures as certain kinds of closure operators, namely, bounded trace operators. In the next subsection we treat some of the basic ideas underlying bounded trace operators, before turning to a presentation of the model.

4.1 The basic model

Trace operators. In general (provided that that the underlying constraint system is expressive enough, see Section 3), a finite closure operator can be represented as the parallel composition of a finite set of finite sequences of asks and tells, where a sequence $a_1!b_1 \star \ldots a_n!b_n\star$ (called a *trace*) is thought of as representing the closure operator $a_1 \rightarrow (b_1 \wedge (a_2 \rightarrow b_2 \ldots (a_n \rightarrow b_n) \ldots)$.[16]

A *trace operator* over a finitary lattice E is, intuitively, a closure operator that can be represented by a *single* (possibly infinite) ask/tell sequence. The characterizing property of a trace operator f is that if $S \subseteq E$ is a set of elements none of which are fixed points of f, then neither is $\sqcap S$ (provided that it exists):

Definition 4.1 A trace operator over a finitary lattice E is a closure operator f over E such that for any $S \subseteq E$, if S is disjoint from f, then $\sqcap S \notin f$ (whenever $\sqcap S$ is defined). Let $\mathcal{T}(E)$ be the set of all trace operators over E. □

Intuitively this definition can be justified as follows. Let d be an arbitrary element in S, and suppose that t is a trace. Then, if d is not a fixed point of t, it should be possible for t to execute some prefix of its actions, including at least one tell action involving a constraint stronger than d, before quiescing. Similarly for any other $e \in S$. Let s be the smallest prefix executed by t in e or d. $d \sqcap e$ will have at least as much information as all the asks in s, so t will be able to execute all of s, including a tell involving a constraint stronger than $d \sqcap e$.

The characteristic condition of a trace operator can be stated much more elegantly as follows. For f a closure operator over a lattice E, define f^{-1}, the *inverse* of f to be the set of elements $(E \setminus f) \cup \{\top_E\}$.

[16] Recall that for $c \in E$ and f a closure operator on e, $c \rightarrow f$ is the closure operator on E with fixed points $\{d \in E \mid d \geq c \Rightarrow d \in f\}$.

Lemma 4.1 *A closure operator $f : E \to E$ is a trace operator iff f^{-1} is a closure operator. If f is a trace operator, then so is f^{-1}.*

f^{-1} is said to be the inverse of f because it is the weakest g satisfying $f \wedge g = f \cap g = \{\top_E\}$. Intuitively, f^{-1} is exactly the sequence of asks and tells that "unzips" f: it asks exactly what f tells and tells exactly what f asks. Consequently, on any input to $f \wedge f^{-1}$, both the sequences can be traversed completely, yielding the final answer $\top_E$. Thus trace operators can be thought of as *invertible* closure operators.

Conversely, it is possible to show that each trace operator can be represented canonically as a sequence of ask/tell actions:

Lemma 4.2 *Every trace operator $f : E \to E$ can be represented by an alternating, strictly increasing sequence of ask/tell actions.*

The basic idea behind the construction of the canonical sequence is quite simple. Let f be a trace operator and $g = f^{-1}$. Then the canonical trace corresponding to f is just:

$$f(\mathtt{true}) \star g(f(\mathtt{true}))!f(g(f(\mathtt{true}))) \star \ldots$$

The following lemma is not difficult to show:

Lemma 4.3 *Let $\mathcal{D} = \langle D, \vdash, \mathtt{Var}, \{\exists_X | X \in \mathtt{Var}\}\rangle$ be a cylindric constraint system. For every $c \in |D|_0$, $Y \in \mathtt{Var}$ and $f \in \mathcal{T}(|D|_0)$, $c, c \to f, c \wedge f, \exists_Y f \in \mathcal{T}(|D|_0)$, where c is the closure operator $\{d \in |D|_0 \mid d \geq c\}$.*

Thus trace operators are closed under almost all the operations of interest to us—except, naturally enough, arbitrary parallel composition.

Bounded trace operators. The failures of a process record a "resting point" of the process, together with information about how to get there. So it would seem as if a failure should be represented as a pair (f, c) where $c \in f$ is the resting point, and f is a trace operator describing the set of ask/tell interactions needed to reach c. Note however, that the only information of interest in f is its behavior on $\downarrow c$. But if $c \in f$, then $f \cap \downarrow c$ is also a trace operator—but on the sub-lattice $\downarrow c$.

Therefore, a *bounded* trace operator (or bto, for short) on a finitary lattice E is defined to be a trace operator on $\downarrow c$, for some $c \in E$. This makes bounded trace operators a special kind of partial trace operators—specifically, those whose range contains a maximal element. (Partial trace operators are just the partial closure operators of Section 3 that are in addition traces.) Let $b\mathcal{T}(E) = \cup_{c \in E} \mathcal{T}(\downarrow c)$ denote the set of all such operators.

Just like any other (partial) trace operator, a bto f is also representable by its range, and its domain of definition is just $\downarrow \tilde{f}$, where $\tilde{f}$ (read "max f") is the greatest fixed point of f. Various operations defined on closure operators are

applicable to btos, with obvious adjustments. Thus, for any constraint c, we shall take the bto corresponding to the imposition of c to be just the bto (whose set of fixed points are) $\{c\}$. Similarly, for a constraint c and trace operator f, with $\tilde{f} \geq c$, the bto $c \rightarrow f$ is just the bto $\{d \leq \tilde{f} \mid d \geq c \Rightarrow d \in f\}$.

However, some additional operations are also of interest over btos. We next discuss operations that reflect the operational notion of extending a sequence of ask/tell interactions with more ask/tell actions.

Let f be a (finite) bto, with canonical sequence of ask/tell actions s, and let $c \geq \tilde{f}$ be any constraint. Define $f.c\star$ (read: "f output extended by c") to be the bto corresponding to the sequence of actions $s.c\star$. It is not hard to find a direct representation of $f.c\star$ in terms of f and c: [17]

$$f.c\star = \{d \leq c \mid d \sqcap \tilde{f} \in f, d \not\geq \tilde{f}\} \cup \{c\}$$

In the following, we shall assume that the expression $f.c\star$ is well-defined even if $c \not\geq \tilde{f}$, and take it to stand for f in such cases. Note that $f.c \star .d\star = f.d\star$, if $c \leq d$.

Similarly, we can define the notion of input extending a bto f with a constraint c by:

$$f.c! = \begin{cases} \{d \leq c \mid d \sqcap \tilde{f} \in f\} & if \tilde{f} \leq c \\ f & o.w. \end{cases}$$

As above, note that $f.c!.d! = f.d!$, if $c \leq d$.

Given the definitions of input- and output-extensions, it is not hard to see that for any sequence of ask/tell actions $s = e_1 e_2 \ldots e_n$, the corresponding closure operator is just $(\ldots((\{true\}.e_1).e_2)\ldots).e_n$ (where we have abused notation by writing $f.e$ for the expression $f.c!$ in case $e \equiv c!$ and for the expression $f.c\star$ in case $e \equiv c\star$).

Let us write $f \sqsubseteq g$ for the case in which f can be thought of as a "prefix" of g, that is, g can be thought of as extending the sequence of interactions with the environment engaged in by f. How can this partial order be expressed directly in terms of (the set of fixed-points of) f and g? Clearly, none of the additional interactions in g can cause g to take on a different value from f at all points in f's domain ($\downarrow \tilde{f}$), *except* possibly at $\tilde{f}$.[18] Therefore, we can define:

$$f \sqsubseteq g \iff \tilde{f} \leq \tilde{g} \text{ and } f = (g \cap \downarrow \tilde{f}) \cup \{\tilde{f}\}$$

As can be verified from the definition, $\sqsubseteq$ is a partial order.

Finally, one more partial order will be of interest in what follows. We say that f *asks more* than g (and write $f \leq g$) if the resting point of both f and

[17] The expression $d \sqcap \tilde{f} \in f$ should be taken to stand for "$d \sqcap \tilde{f}$ is contained in f, provided that it exists".

[18] For an example of a closure operator g which extends f but takes on a different value at $\tilde{f}$ than f, consider the closure operators obtained from the sequences $a!b\star$ and $a!b \star c\star$, for $a \leq b < c$.

g are identical, but f records more contributions from the environment than g. This happens just in case $\tilde{g} = \tilde{f}$ and $\forall x \in \downarrow \tilde{g}.f(x) \leq g(x)$, that is, just in case $\tilde{g} = \tilde{f}$ and $f \supseteq g$.

4.2 The model

Let the set of all observations, $Obs(|D|_0)$, be the set of finite, bounded trace operators on $|D|_0$. A process will be a subset of Obs satisfying certain conditions which we now motivate, following [Jos90] closely.

At any stage of the computation, a process will have engaged in some ask/tell interactions with the store. Subsequently, it may produce some output and then quiesce (perhaps to be activated on more input at a subsequent stage) or it may engage in an infinite sequence of actions, without requiring any input from the environment to progress (perhaps producing more and more output as it progresses). We will model a process as divergent if it can quiesce in infinitely many ways (or output forever) or if it causes the store to become inconsistent.[19] (Thus we are requiring that processes be finitely nondeterminate.) If F is the set of failures of such a process, its divergences can then be defined as:

$$\begin{aligned} dF = \ & \{f \mid \{c \mid f.c\star \in F\} \text{ is infinite}\} \\ & \cup \{f \mid f.\texttt{false}\star \in F\} \end{aligned}$$

Each of these situations is considered undesirable, and we are not concerned about detailed modelling of the process once it has become divergent. Thus such a process is treated as "chaotic", as being able to exhibit any behavior whatsoever. Further, we require that the set of possible behaviors of a process contain *all* its possible behaviors, especially its diverging ones. Thus the first condition we impose on a process F is:

$$edF \subseteq F \tag{12}$$

where for any $S \subseteq Obs$, eS is the set $\{s \sqsupseteq t \mid t \in S\}$ of extensions of S.

Note that $f.c\star \in dF$ implies $f \in dF$. Thus the last action in a sequence of ask/tell interactions constituting a minimal divergence must be an ask action. In other words, a divergence characterizes those inputs from the environment that are *undesirable*, that can cause the process to break.

From the definition, it should be clear that d distributes through finite unions and arbitrary intersections of arbitrary sets of observations. Also, the divergences of a process can be characterized rather nicely:

Lemma 4.4

$$f \in dF \iff \forall g \sqsupseteq f.g \in F \iff f.\texttt{false}\star \in F$$

[19] As discussed in the introduction, it is quite reasonable in this set-up to regard the process that produces the inconsistent store as divergent. It is possible to give a minor variation of the current treatment which distinguishes the process that diverges from the process that tells `false`, but this is outside the scope of this paper.

The next few conditions are best motivated by considering the *traces* of a process. A trace of a process is just a sequence of ask/tell interactions that the process many engage in (without necessarily reaching a quiescent state). But the traces of a process can be recovered in a simple way from its failures: they are just the observations which can be output-extended to obtain a failure:

$$tF = \{f \mid \exists c.f.c\star \in F\}$$

Clearly, $F \subseteq tF$ and t distributes through arbitrary unions of sets of observations.

We require that if a process exhibits a trace, then it should be possible for it to exhibit a prefix of the trace as well—this is inherent in the very idea of a trace:

$$g \sqsubseteq f \in tF \Rightarrow g \in tF \tag{13}$$

We also require that every process should have *some* behaviors, hence a nonempty set of failures. Given Condition 13, this is equivalent to stating that the "empty" bto $\texttt{true}! = \texttt{true}\star = \{\texttt{true}\}$ be a trace of every process:

$$\{\texttt{true}\} \in tF \tag{14}$$

Since `cc` processes are asynchronous, the environment can never be prevented from adding constraints to the store. Therefore, it should be possible to extend every sequence of interactions that a process may have with its environment with an input action: (the *receptiveness* condition):

$$f \in tF \Rightarrow f.c! \in tF \tag{15}$$

It is not hard to show that for any chain of processes $F_1 \supseteq F_2 \ldots$, $t \cap_{i \geq 1} F_i = \cap_{i \geq 1} tF_i$.

We require one final condition on processes. If a process can engage in a sequence of actions recorded by a bto f before quiescing, then it can engage in the same sequence of actions even if at some or all stages the environment were to supply more input than the minimum required by the process to engage in f. Thus we require that the failures of a process be closed under the "ask more" relationship:

$$g \leq f \in F \Rightarrow g \in F \tag{16}$$

In essence, this condition represents the monotonic nature of the basic ask and tell actions.

Now we are ready to define:

Definition 4.2 A (nondeterminate) process is a subset of $Obs(|D|_0)$ satisfying Conditions 12–16. Let `NProc` be the set of all such subsets. □

The following lemma establishes that the convergences of a process already contain enough information to generate its divergences. (The converse is not true.) For F a process, define iF, the input extensions of F to be the set $\{f.c! \mid f \in F\}$, and cF, the *convergences* of F to be the set $F \setminus dF$.

Lemma 4.5 $dF = e((icF \cup \{\{\texttt{true}\}\}) \setminus tcF)$

Essentially, any input extension of a convergent trace of a process must have an output extension that is a failure of the process (Condition 15); if this output extension is not a convergence, it must be a divergence and so must its extensions. Conversely, a divergence of a process must have a prefix which input-extends a convergence and is not itself a convergent trace; otherwise the process is chaotic, and every bto is a divergence.

Partial order on processes. Processes in specification-orien- ted semantics are usually ordered by the so-called *nondeterminism* ordering:

$$F \sqsubseteq G \stackrel{d}{=} F \supseteq G$$

which corresponds to the intuition that a process is "better" than another if it is more deterministic. The completely undefined process is the chaotic process, which can exhibit all possible behaviors: as more and more information about a process is generated, more and more behaviors get ruled out.

However, in many senses, this ordering is more liberal than desired, as discussed by Roscoe in [Ros88]. For example, one way in which a process G can improve a process F is by dropping some *convergent* behavior of F. This sort of capability is not manifested by any `cc` combinator (or, indeed, any CSP combinator), and we find it more convenient to adopt instead the *divergence* ordering proposed by Roscoe. In this ordering G is "better" than F iff it diverges at fewer places than F, and the convergent behaviors of F are preserved in G. More precisely, the partial order is:

$$F \leq G \stackrel{d}{=} cF \subseteq cG \subseteq F$$

It is easy to see from the definition that $F \leq G$ implies $F \sqsubseteq G$. Furthermore, the least element in the partial order is Obs, and limits of increasing chains are given by intersection. In fact, if $F_1 \leq F_2 \leq \ldots$ is an increasing chain with lub $F = \cap_{i \geq 1} F_i$, we have $cF = \cup_{i \geq 1} cF_i$ and $dF = \cap_{i \geq 1} dF_i$.

Theorem 4.6 $\langle \texttt{NProc}, \leq \rangle$ *is a complete partial order.*

4.3 Process algebra

In this section, we define various processes in and combinators on `NProc`, including **div**, the process that immediately diverges, the tell (of finite constraints),

Syntax.

$$P ::= D.A$$
$$D ::= \epsilon \mid p(X) :: A \mid D.D$$
$$A ::= c \mid A \wedge A \mid \exists X A \mid p(X) \mid c_1 \rightarrow A_1 \,\square\, \ldots \,\square\, c_j \rightarrow A_j$$

Auxiliary Definitions.

$$\begin{aligned}
dF = \;& \{f \mid \{c \mid f.c\star \in F\} \text{ is infinite}\} \\
& \cup \{f \mid f.\texttt{false}\star \in F\} \\
tF = \;& \{f \mid \exists c. f.c\star \in F\} \\
F \| G = \;& \{f \sqcap g \in Obs \mid \tilde{f} = \tilde{g}, f \in F, g \in G\} \\
X \hat{} F = \;& \\
& \{g \in Obs \mid g \leq \downarrow d \sqcap (\exists X(f.\texttt{false}!)), \exists_X d = \exists_X \tilde{f}, f \in F\} \\
\exists_X(d_{XY} \sqcup F) = \;& \{\{\exists_X(d_{XY} \sqcup c) \mid c \in f\} \mid f \in F\}
\end{aligned}$$

Semantic Equations.

$$\begin{aligned}
\mathcal{A}(c)e = \;& \{f \mid \tilde{f} \sqcap \uparrow c \subseteq f, c \leq \tilde{f}\} \\
& \cup \{g \sqsupseteq f \mid \tilde{f} \sqcap \uparrow c \subseteq f, c \sqcup \tilde{f} = \texttt{false}\} \\
\mathcal{A}(\square_{j \in J}(c_j \rightarrow A_j))e = \;& \\
& \{f \in \mathcal{A}(A_j)e \mid f = c_j \rightarrow f, \tilde{f} \geq c_j, j \in J\} \\
& \cup \{\downarrow d \mid \forall j \in J. d \not\geq c_j\} \\
\mathcal{A}(A \wedge B)e = \;& \mathcal{A}(A)e \| \mathcal{A}(B)e \cup ed(t\mathcal{A}(A)e \| t\mathcal{A}(B)e) \\
\mathcal{A}(\exists X A)e = \;& (X \hat{} \mathcal{A}(A)e) \cup \{g \leq f \mid f \in ed(X \hat{} \mathcal{A}(A)e)\} \\
\mathcal{A}(p(X))e = \;& \exists_\alpha(d_{\alpha X} \sqcup e(p)) \\
\mathcal{E}(\epsilon)e = \;& e \\
\mathcal{E}(p(X) :: A.D) = \;& \mathcal{E}(D)e[p \mapsto \exists_X(d_{\alpha X} \sqcup \mathcal{A}(A)e)] \\
\mathcal{P}(D.A) = \;& \mathcal{A}(A)(\underline{\texttt{fix}}\, \mathcal{E}(D))
\end{aligned}$$

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

Table 2: Denotational semantics for the Ask-and-Tell nondeterminate cc languages

parallel composition, nondeterminate choice and hiding combinators. The syntax of the programming language is given in Table 2, where the semantic definitions, to be discussed below, are also summarized. As before, we also simultaneously define the operational semantics of the language and assume that procedures take exactly one variable as a parameter and that no program calls an undefined procedure.

Chaos. The chaotic process can do anything whatsoever.

$$\mathbf{div} = Obs$$

Clearly, $d(\mathbf{div}) = t(\mathbf{div}) = Obs$. Operationally, such an agent is always willing to progress in any store. This progress affects neither the store nor the agent's subsequent behavior:

$$\mathbf{div} \xrightarrow{(d,d)} \mathbf{div} \tag{17}$$

Tells. Consider a process which immediately terminates, after augmenting the store with some constraint $c \in |D|_0$. Let us call such a process c. The resting points of such a process are clearly all stores $e \geq c$. To reach this resting point, the process can at most add c to the store. It is not hard to see that a bto f satisfies the condition that for all inputs x in its domain, $f(x) \leq x \sqcup c$ iff $\tilde{f}\sqcap \uparrow c \subseteq f$. When does such a process diverge? It must diverge iff it can engage in some sequence of interactions with the store (in which its output is bounded by c), after which it reaches a state in which if it were to output c, it would reach `false`:

$$\begin{aligned} c = \ & \{f \mid \tilde{f}\sqcap \uparrow c \subseteq f, \tilde{f} \geq c\} \\ & \cup e\{f \mid \tilde{f}\sqcap \uparrow c \subseteq f, \tilde{f} \sqcup c = \texttt{false}\} \end{aligned}$$

It is easy to work through the definitions and establish that

$$\begin{aligned} d(c) &= e\{f \mid \tilde{f}\sqcap \uparrow c \subseteq f, \tilde{f} \sqcup c = \texttt{false}\} \\ t(c) &= \{f \mid \tilde{f}\sqcap \uparrow c \subseteq f\} \cup d(c) \end{aligned}$$

and that c (as defined above) is a process.

The relevant axiom for the transition relation for these agents is the same as in the determinate case (Axiom 4).

Dependent choice. Consider the process $F \equiv \ \square_{j\in J}(c_j \rightarrow F_j)$. It has two kinds of resting points: first, the resting points d that arise because for no $j \in J$ is $d \geq c_j$, and secondly, the resting points of each F_j which are stronger than the corresponding c_j. Furthermore, the btos generating the first kind of resting point are simple: they are of the form $d! =\downarrow d$, since no output is produced by the process before it quiesces in d. On the other hand, the path followed by

F in reaching a resting point of F_j stronger than c_j is the path that F_j would have followed given that the environment is willing to supply at least c_j, that is paths $f \in F_j$ such that $f = c_j \rightarrow f$. This leads us to the definition:

$$\begin{array}{rl} \Box_{j \in J}(c_j \rightarrow F_j) = & \{f \in F_j \mid f = c_j \rightarrow f, c_j \leq \tilde{f}, j \in J\} \\ & \cup \{\downarrow d \mid \forall j \in J.d \not\geq c_j\} \end{array}$$

As can be calculated, the divergences and traces of F are:

$$\begin{array}{l} d(\Box_{j \in J}(c_j \rightarrow F_j)) = \\ \quad \{f \in dF_j \mid f = c_j \rightarrow f, c_j \leq \tilde{f}, j \in J\} \\ t(\Box_{j \in J}(c_j \rightarrow F_j)) = \\ \quad \{f \in tF_j \mid f = c_j \rightarrow f, c_j \leq \tilde{f}, j \in J\} \\ \quad \cup \{\downarrow d \mid \forall j \in J.d \not\geq c_j\} \end{array}$$

The combinator is monotone and continuous in each of its process arguments.

Two special cases of this operator are worth singling out. In case the index-set is singleton, dependent choice is not a form of choice at all and reduces to just the ask-combinator. That is, $c \rightarrow A$ is just dependent choice in which only one conditional is given. In terms of denotations, we get:

$$c \rightarrow F = \{f \in F \mid f = c \rightarrow f, c \leq \tilde{f}\} \cup \{\downarrow d \mid d \not\geq c\}$$

Note that for such agents, the transition Axiom 11 reduces to just Axiom 5 (Section 3).

Similarly, blind unconditional choice can also be expressed. Consider the binary combinator $\sqcap$ defined such that $F \sqcap G$ can behave either like F or like G. The decision can be made arbitrarily, even at compile-time. Thus the failures of $F \sqcap G$ should be precisely the failures of F or the failures of G. As can easily be checked, $F \cup G = \texttt{true} \rightarrow F \,\Box\, \texttt{true} \rightarrow G$ as well. Therefore, $F \sqcap G$ can be defined as $\texttt{true} \rightarrow F \,\Box\, \texttt{true} \rightarrow G$. Clearly, blind choice is idempotent, associative and commutative and has **div** as a zero element. Operationally, an agent built from blind choice satisfies the axioms:

$$\begin{array}{l} A \sqcap B \xrightarrow{(d,d)} A \\ A \sqcap B \xrightarrow{(d,d)} B \end{array} \tag{18}$$

Parallel composition. What are the resting points of $F \wedge G$? Clearly, if c is a resting point of F and of G, then it is a resting point of $F \wedge G$. The path followed to this resting point by $F \wedge G$ can be any parallel composition of the paths followed by F and by G. Therefore, each failure in the set $F||G$ is going to be a failure of $F \wedge G$, where

$$F||G = \{f \cap g \in Obs \mid \tilde{f} = \tilde{g}, f \in F, g \in G\}$$

But what are the divergences of $F \wedge G$? The divergences of $F \wedge G$ arise not only from the divergences of F and of G, but also from the possibility that the two agents may engage in an infinite sequence of interactions with each answering the others asks, without demanding input from the environment at any stage. There will be no bto in $F||G$ corresponding to such "mutual feedback" because there is no common resting-point on this execution branch.

Capturing these possibilities for a `cc` language built over an arbitrary constraint system seems rather subtle. A simple formulation is possible, however, for finitary constraint systems, that is, constraint systems in which a finite element dominates only finitely many finite elements. In this case we can show:

Lemma 4.7 *If $\mathcal{D}$ is a finitary constraint system, then for every $F \in \mathtt{NProc}(\mathcal{D})$, $dF = dtF$.*

This suggests that to determine the divergences of $F \wedge G$, it is sufficient to determine the divergences of the traces of $F \wedge G$. But this is easy: a trace of $F \wedge G$ is just a trace of F running in parallel with a trace of G, and hence the divergent traces are just $ed(tF||tG)$. We thus get:

$$F \wedge G = F||G \cup ed(t(F)||t(G))$$

From these we can caculate:

$$d(F \wedge G) = ed(tF||tG)$$
$$t(F \wedge G) = (tF||tG) \cup ed(tF||tG)$$

Proving continuity of this operator requires some care. The basic issue is to show that $d(tF||tG)$ is continuous in its arguments.

The operational transition rule for agents built with $\wedge$ is the same as in the deterministic case (Axiom 6).

Projection. d is a resting point of $\exists XF$ iff when F is initiated in $(\exists_X d)$, it reaches a resting point e such that the only new information in e (over d) is on X; that is, such that $(\exists_X e) = (\exists_X d)$. Therefore d is a resting point of $\exists XF$ iff there is an $f \in F$ such that $\exists_X \tilde{f} = \exists_X d$. The route taken by $(\exists XF)$ to reach d from `true` must be the route prescribed by $\exists Xg$, where g is obtained from f by extending it to be a closure operator over $|D|_0$, restricted to $\downarrow d$. Thus define:

$$X\hat{}F =$$
$$\{g \in Obs \mid g \leq \downarrow d \cap (\exists X(f.\mathtt{false}!)), \exists_X d = \exists_X \tilde{f}, f \in F\}$$

Now the failures of $\exists XF$ are:

$$\exists XF = X\hat{}F \cup \{g \leq f \mid f \in ed(X\hat{}F)\}$$

The divergences and traces for this process can be shown to be:

$$d(\exists XF) = \{g \leq f \mid f \in ed(X\hat{}F)\}$$
$$t(\exists XF) = X\hat{}tF \cup \{g \sqsubseteq f \mid f \in d(X\hat{}F)\}$$

This operator is monotone and continuous in its argument. The transition relation for $\exists XA$ is the same as in the determinate case; the transition relation must therefore be extended to agents with an internal store.

Recursion. Recursion is handled in the usual way, by taking limits of the denotations of all syntactic approximants, since the underlying domain of processes is a cpo and all the combinators are continuous in their process arguments. The diagonal elements are used to effect parameter passing (see Table 2). Operationally, procedure calls are handled as in the determinate case.

4.4 Operational Semantics

The operational semantics associates with every program and every initial store the set of all possible outputs obtainable from the store, with the caveat that if the process diverges or produces `false`, then every output is deemed observable. Here the statement P has a (c,d)-sequence means there exists a terminal c-sequence for P with final store d:

Definition 4.3 For any program P define

$$\mathcal{O}(P) = \begin{cases} Obs & \text{if } P \text{ has an infinite } \texttt{true}\text{-sequence} \\ Obs & \text{if } P \text{ has a } (\texttt{true}, \texttt{false})\text{-sequence} \\ \{d \mid P \text{ has a } (\texttt{true}, d)\text{-sequence}\} & \text{otherwise} \end{cases}$$

□

Relationship with the denotational semantics. We make the connection between the operational semantics and the denotational semantics via the following theorems. The proofs are omitted in this version.

Theorem 4.8 (Adequacy) $\mathcal{O}(P) = \{d \mid \{d\} \in \mathcal{P}(P)\}$

That is, the results obtained by executing a program are identical to the resting points of the program obtained from the store `true`. Note that this is a weaker correspondence than in the determinate case, when the operational semantics was identical to the denotational semantics. The following full abstraction proof uses the notion of context previously defined for determinate programs.

Theorem 4.9 (Full abstraction) $\mathcal{P}(P) = \mathcal{P}(Q)$ *iff for all contexts* $\mathcal{C}[\bullet]$, $\mathcal{O}(\mathcal{C}[P]) = \mathcal{O}(\mathcal{C}[Q])$.

If the denotations of two programs P and Q are different, then there will be a $\leq$-maximal bto f in one and not in the other. It can be shown that the sequence s corresponding to such an f can be expressed in the language, which

implies that the sequence s^{-1} corresponding to f^{-1} can also be expressed in the language. But then the finite bto $s^{-1} \wedge \bullet$ is a context which distinguishes the two programs. Let $F = \mathcal{P}(P)$, $G = \mathcal{P}(Q)$. Assume without loss of generality that $f \in F$ and $f \notin G$. There are two cases: $f \in dF$ and $f \in cF$. If $f \in dF$ then $s^{-1} \wedge F$ will diverge, and $\mathcal{O}(s^{-1} \wedge F) = |D|_0$. $f \notin G$ implies $s^{-1} \wedge G$ will not diverge, and therefore $\texttt{false} \notin \mathcal{O}(s^{-1} \wedge G)$. If $f \in cF$, then $\tilde{f} \in \mathcal{O}(s^{-1} \wedge F)$. If $\tilde{f} \in \mathcal{O}(s^{-1} \wedge G)$ then f must be a convergence of G, which violates the assumption.

4.5 Relationship between the nondeterminate and determinate semantics

We have only one set of transition rules for the determinate combinators, and the same notion of observation for the determinate and nondeterminate semantics. Therefore, the operational semantics for the determinate language and the determinate subset of the nondeterminate language are the same. Because both the determinate ($\mathcal{A}_D$) and nondeterminate ($\mathcal{A}_N$) denotational semantics are fully abstract with respect to the corresponding operational semantics, there should be some relationship between $\mathcal{A}_N(P)$ and $\mathcal{A}_D(P)$. Consider the two determinate agent-equivalence classes C_D and C_N induced by $\mathcal{A}_D$ and $\mathcal{A}_N$, respectively. Because the nondeterminate language has more contexts with which to tell apart agents than the determinate language, C_D should be a coarsening of C_N, and we should therefore be able to recover a determinate program's determinate denotation from its nondeterminate denotation.

Definition 4.4 An element $F \in \texttt{NProc}$ is *determinate* iff

1. $f \in cF$, $F \in \texttt{NProc}$ and $c > \tilde{f}$ implies $f.c\star \notin tF$
2. $f \in cF$, $g \in tF$ and $\tilde{g} = \tilde{f}$ implies $g \in cF$.

Let $\texttt{DNProc}$ be the subset of determinate processes of $\texttt{NProc}$. □

Henceforth when we say *determinate*, we mean that we have an agent in $\texttt{NProc}$ that satisfies the determinacy condition and not an element of the syntactic class of determinate processes. We, of course, would like the denotation of agents built from the determinate combinators to be determinate:

Theorem 4.10 *$\mathcal{A}_N(A)$ is determinate if A is constructed using the determinate combinators.*

We are now ready to define DN, which associates with $F \in \texttt{DNProc}$ a corresponding element in $\texttt{DProc}$, the domain used for the denotational semantics of determinate agents in Section 3.

Definition 4.5 $DN(F) = \{\tilde{f} \mid f \in cF\} \cup \{false\}$, for $F \in \texttt{DNProc}$. □

Theorem 4.11 *$DN(\mathcal{A}_N(A)) = \mathcal{A}_D(A)$ for all agents A constructed from the determinate operators.*

We prove this theorem by first showing that $DN(\mathcal{A}_N(A))$ is a closure operator. We then show by structural induction that the theorem holds for finite agents built using the determinate combinators. We prove the theorem for recursively defined agents by showing that DN is monotone and continuous.

5 Conclusion and Future work

This paper presents a comprehensive treatment of the specifi- cation-oriented approach to the semantics of programs written in concurrent constraint programming languages. This treatment includes programs built using recursion. By formalizing a general notion of constraint system, we cleanly separate the semantics of the programming language combinators from the semantics of the underlying data domain. This separation allows us to uniformly address the semantics of a wide variety of concurrent constraint programming languages with a single general framework. These languages include, among others, the concurrent logic programming languages and Kahn data-flow networks.

Our work brings into sharp focus the semantic complexity caused by having nondeterminacy in the cc languages. The determinate semantics need only record the stores at which a process quiesces – there is no need to maintain any intermediate process state information. The nondeterminate semantics, on the other hand, must record both the stores in which a process may quiesce, and, for each such store, the possible computation paths to that store. It is interesting that finitariness plays a key role in the nondeterminate semantics but not in the determinate semantics.

We make the connection between the determinate semantics and the nondeterminate semantics by defining an operator that extracts the determinate denotation of a program built with the determinate combinators from its nondeterminate denotation. We also present an equational axiomatization that is complete for finite programs built with the determinate combinators.

This paper also presents a single transition system for both the determinate and nondeterminate languages. This transition system uses diagonal elements and local stores to eliminate messy variable renaming operations.

There are many directions for future research. These include foundational concerns, such as are addressed here, implementation issues and applications. We intend to pursue all these issues in the coming months. In this section we only mention the semantic issues.

We have by no means exhausted the range of interesting combinators that are available in the determinate cc languages. For example, the *glb* operator on agents is also available, and provides a sort of determinate "disjunction". Some of these operators will be treated in the complete version of this paper. A useful

line of investigation is to try to characterize "all sensible combinators" that one may use. Here general results from category theory may help.

There are a variety of different semantics corresponding to different notions of observations. We would like to develop a semantics for the indeterminate case that is not based on viewing divergence as chaos. This would be like Plotkin's powerdomain treatment of indeterminate imperative languages [Plo76]. In subsequent work we plan to develop proof systems for safety and liveness properties of cc programs based on these models. In a related paper we are developing the closely related safety model and an axiomatization of equality for it.

We also believe that it is possible to develop a theory of higher-order determinate cc programming languages. There are interesting connections to be made with other theories of higher-order concurrent processes [BB90,JP90,Mil90] and also with classical linear logic. It appears that concurrent constraint languages may be related to the proof nets introduced by Girard in his discussion of the proof theory of linear logic. If this connection were successful it would exhibit concurrent constraint programs as arising from linear logic via a Curry-Howard isomorphism.

Acknowledgements This research was supported in part by DARPA contract N00014-87-K-0828, NSF grant CCR-8818979 to Cornell University and an NSERC grant to McGill University. We gratefully acknowledge discussions with Seif Haridi, Tony Hoare, Radha Jagadeesan, Mark Josephs, Ken Kahn, John Lamping, Keshav Pingali and Gordon Plotkin. The debt our treatment owes to Mark's development of receptive processes should be clear to anyone who has read his paper. None of them should be held responsible for any remaining errors.

References

[ANP89] Arvind, Rishiyur Nikhil, and Keshav K. Pingali. I-structuers: data-structures for parallel computing. *ACM Transactions on Principles of Programming Languages*, 11(4):598–632, October 1989.

[BB90] G. Boudol and G. Berry. The chemical abstract machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, 1990.

[CM88] Mani Chandy and Jay Misra. *Parallel Program Design—A foundation.* Addison Wesley, 1988.

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

[dBP90b] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. June 4 1990.

[DL86] Doug DeGroot and Gary Lindstrom, editors. *Logic Programming: Functions, Relations and Equations*. Prentice Hall, 1986.

[FT89] Ian Foster and Steve Taylor. *Strand: New concepts in parallel programming*. Prentice Hall, 1989.

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

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir89] J.-Y. Girard. *Proofs and Types*, volume 7 of *Cambridge tracts in Theoretical Computer Science*. Cambridge University Press, 1989. Translated and with appendices by Y. Lafont and P. Taylor.

[GKK+80] G.Gierz, K.H.Hoffman, K.Keimel, J.D.Lawson, M.Mislove, and D.S.Scott, editors. *A compendium of continuous lattices*. Springer-Verlag Berlin Heidelberg New York, 1980.

[GL90] M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics of concurrent constraint logic programs. Technical report, University of Pisa, 1990.

[GMS89] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *North American Logic Programming Conference*. MIT Press, October 1989.

[HMT71] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras (Part I)*. North Holland Publishing Company, 1971.

[Hoa89] C.A.R. Hoare. A theory of conjunction and concurrency. Oxford PRG, May 1989.

[Hoa90] C.A.R. Hoare. Let's make models. In *Proceedings of CONCUR 90*, August 1990.

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

[Jos90] Mark B. Josephs. Receptive process theory. Technical report, Programming Research Group, Oxford University, July 1990.

[JP90] R. Jagadeesan and P. Panangaden. A domain-theoretic model of a higher-order process calculus. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 181–194. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

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

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proceeedings of IFIP Congress 74*, pages 471–475., August 1974.

[KYSK88] S. Kliger, E. Yardeni, E. Shapiro, and K. Kahn. The language **fcp(:,?)**. In *Conference on Fifth Generation Computer Systems*, December 1988.

[Lev88] Giorgio Levi. Models, unfolding rules and fixpoint semantics. In *Proceeedings of the Fifth International Conference and Symposium on Logic Programming, Seattle*, pages 1649–1665, August 1988.

[Lin85] Gary Lindstrom. Functional programming and the logical variable. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 266–280, January 1985.

[LS86] J. Lambek and P. Scott. *An introduction to higher-order categorical logic*, volume 7 of *Studies in Advanced Mathematics*. Cambridge University Press, 1986.

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

[Mil90] R. Milner. Functions as processes. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 167–180. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

[MPW89] R. Milner, J. G. Parrow, and D. J. Walker. A calculus for mobile processes. LFCS Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[MR97] M. Makkai and G. Reyes. *First order categorical logic*, volume 611 of *Lecture Notes in Mathematics*. Springer-Verlag, 197.

[OH86] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

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

[Ros88] A. W. Roscoe. An alternative order for the failures model. Technical Report Technical Monograph PRG-67, Programming Research Group, Oxford University, July 1988.

[Sar85] Vijay A. Saraswat. Partial correctness semantics for **cp(↓,|, &)**. In *Proceedings of the FSTTCS Conference*, number 206, pages 347–368. Springer-Verlag, December 1985.

[Sar88] Vijay A. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In *Proceedings of LP 88*. MIT Press, August 1988.

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

[Sco76] Dana S. Scott. Data types as lattices. *SIAM*, 5(3):522–587, 1976.

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

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

[SPRng] Vijay A. Saraswat, Prakash Panangaden, and Martin Rinard. What is a constraint? Technical report, Xerox PARC, forthcoming.

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

[Tar56] A. Tarski. *Logics, semantics and meta-mathematics*. Oxford University Press, 1956. Translated by J.H. Woodger.