in the resting points of the process to uniquely determine its associated function. This property highlights the semantic simplicity of the Ask-and-Tell communication and synchronization mechanism.

Let f be the operator on $|D|_0$ corresponding to a given process. Now, the only way in which this process can affect the store is by adding more information to it. Therefore, f must be *extensive*:

$$\forall c. c \leq f(c) \tag{1}$$

Second, the store is accessible to the process as well as its environment: therefore if on input c a process produces output d and halts, it must be the case that d is a resting point, that is, on input d the process cannot progress further (because if it could, then it wouldn't have stopped in d). That is, f must be idempotent:

$$\forall c. f(f(c)) = f(c) \tag{2}$$

Finally, consider what happens to the output of such a function when the information content of the input is increased. If the invocation of the function corresponds to the imposition of a constraint, then as information in the input is increased, information in the output should not decrease. Such a function is called *monotone*:

$$\forall c, d. c \leq d \Rightarrow f(c) \leq f(d) \tag{3}$$

An operator over a partial order that is extensive, idempotent and monotone is called a *closure operator* (or, more classicaly, a *consequence* operator, [Tar56]). Closure operators are extensively studied in [GKK+80] and enjoy several beautiful properties which we shall exploit in the following. Within computer science, continuous closure operators have been used in [Sco76] to characterize data-types.

We list here some basic properties. The most fundamental property of a closure operator f over a lattice E is that it can be represented by its range $f(E)$ (which is the same as its set of fixed points). f can be recovered from $f(E)$ by mapping each input to the least element in $f(E)$ above it. This is easy to see: by extensiveness, $f(c)$ is above c; by idempotence, $f(c)$ is a fixed-point of f, and by monotonicity, it is the least such fixed-point. This representation is so convenient that in the following, we shall often confuse a closure operator with its range, writing $c \in f$ to mean $f(c) = c$. In fact, a subset of E is the range of a closure operator on E iff it is closed under glbs of arbitrary subsets (whenever such glbs exist in the lattice). Thus, the range of every closure operator is non-empty, since it must contain $\mathtt{false} = \sqcap\emptyset$.

Partial order. For a closure operator f over $|D|_0$, let df be the *divergences* of f, that is, those inputs in which the process diverges. As discussed above, the divergences are exactly those constraints which are mapped to $\mathtt{false}$ by f, that is, $f^{-1}(\mathtt{false})$. The partial order on determinate processes of interest to us will be based partly on the processes' divergences. The intention is that a process f can be improved to a process g iff the divergences of g are contained in those of f and at every point in the convergences of f, f and g take on identical values. In terms of fixed points, this yields:

Definition 3.1 The divergence order on closure operators over $|D|_0$ is given by:

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

□

The bottom element of this partial order is $\{\mathtt{false}\}$ which diverges everywhere. It is not hard to see that this order is complete, with limits of chains given by unions of the set of fixed-points.

3.1 Process Algebra

In this section we develop a simple language, the determinate cc language, for expressing the behavior of concurrent determinate constraint processes. We consider agents constructed from tells of finite constraints, asks of finite constraints, hiding operators, parallel composition and procedure calls. Throughout this section we shall assume some fixed cylindrical constraint system (with diagonal elements) D. As usual, $|D|$ denotes this constraint system's set of elements, while $|D|_0$ denotes its set of finite elements.

We also define a quartic *transition* relation

$$\longrightarrow \subseteq Env \times (|D|_0 \times |D|_0) \times A \times A$$

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

The syntax and semantics of the determinate language are given in Table 1. We discuss these semantic definitions in this section. For purposes of exposition we assume that procedures take exactly one variable as a parameter and that no program calls an undefined procedure. We also systematically confuse the syntactic object consisting of a finite set of tokens from D with the semantic object consisting of this set's closure under $\vdash$.

Tells. The process c augments its input with the finite constraint c. Thus it behaves as the operator $\lambda x. c \sqcup x$, which in terms of fixed points, is just:

$$c = \{d \in |D|_0 \mid d \geq c\}$$

The operational behavior of c is described by the transition axiom:

$$c \xrightarrow{(d, c \sqcup d)} \mathtt{true} \quad (c \neq \mathtt{true}) \tag{4}$$

corresponding to adding the information in c to the shared constraint in a single step.[13]

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

$$c \rightarrow f = \{d \in |D|_0 \mid d \geq c \Rightarrow d \in f\}$$

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

Syntax.

$$P ::= D.A$$
$$D ::= \epsilon \mid p(X) :: A \mid D.D$$
$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid \exists X A \mid p(X)$$

Semantic Equations.

$$\mathcal{A}(c)e = \{d \in |D|_0 \mid d \geq c\}$$
$$\mathcal{A}(c \rightarrow A)e = \{d \in |D|_0 \mid d \geq c \Rightarrow d \in \mathcal{A}(A)e\}$$
$$\mathcal{A}(A \wedge B)e = \{d \in |D|_0 \mid d \in \mathcal{A}(A)e \wedge d \in \mathcal{A}(B)e\}$$
$$\mathcal{A}(\exists X A)e = \{d \in |D|_0 \mid \exists c \in \mathcal{A}(A)e . \exists_X d = \exists_X c\}$$
$$\mathcal{A}(p(X))e = \exists_\alpha(d_{\alpha X} \sqcup e(p))$$
$$\mathcal{D}(\epsilon)e = e$$
$$\mathcal{D}(p(X) :: A.D) = \mathcal{D}(D)e[p \mapsto \exists_X(d_{\alpha X} \sqcup \mathcal{A}(A)e)]$$
$$\mathcal{P}(D.A) = \mathcal{A}(A)(\ \underline{\texttt{fix}}\ \mathcal{D}(D))$$

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 1: Denotational semantics for the Ask-and-Tell Determinate cc languages

The ask operation is monotone and continuous in its process argument. It satisfies the laws:

$$\begin{array}{ll} (L1) & c \rightarrow d = c \rightarrow (c \wedge d) \\ (L2) & c \rightarrow \mathbf{true} = \mathbf{true} \\ (L3) & c \rightarrow d \rightarrow A = (c \sqcup d) \rightarrow A \\ (L4) & \mathbf{true} \rightarrow A = A \end{array}$$

The rule for $c \rightarrow A$ is:

$$c \rightarrow A \xrightarrow{(d,d)} A \quad \text{if} \quad d \geq c \tag{5}$$

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

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

It is straightforward to verify that this operation is well-defined, and monotone and continuous in both its arguments.

While the argument given above is quite simple and elegant,[14] it hides issues of substantial complexity. The basic property being exploited here is the *restartability* of a determinate process. Suppose an agent A is initiated in a store c, and produces a constraint d before quiescing, leaving a "residual agent" B to be executed. To find out its subsequent behavior (e.g., to find out what output it would produce on a store $e \geq d$), it is *not* necessary to maintain any explicit representation of B in the denotation of A. Rather, the effect of B on input $e \geq d$ can be obtained simply by running the *original* program A on e! Indeed this is the basic reason why it is possible to model a determinate process accurately by just the set of its resting points.

[14]And should be contrasted with most definitions of concurrency for other computational models which have to fall back on some sort of interleaving of basic actions.

As we shall see in the next section, this restartability property is not true for nondeterminate processes. Indeed, we cannot take the denotation of a process to be a function (nor even a relation) from finite stores to finite stores; rather it becomes necessary to also preserve information about the *path* (that is, the sequence of ask/tell interactions with the environment) followed by the process in reaching a resting point.

From this definition, several laws follow immediately. Parallel composition is commutative, associative, and has an identity element.

$$\begin{array}{ll} (L5) & A \wedge B = B \wedge A \\ (L6) & A \wedge (B \wedge C) = (A \wedge B) \wedge C \\ (L7) & A \wedge \mathbf{true} = A \end{array}$$

Telling two constraints in parallel is equivalent to telling the conjunction. Prefixing distributes through parallel composition.

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

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.

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

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 $\exists X B$, 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 $d' \in |D|$, $c \sqcup \exists_X d' \geq c'$ iff $\exists_X c \sqcup \exists_X d' \geq d$.

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 = Λ → 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 $A = \mathrm{true}$ 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 $\exists X B$ 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 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:

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

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: