Information and Computation **146**, 138–180 (1998)
Article No. IC982738

A Fully Abstract Semantics for

Sven-Olof Nyström

Computing Science Department, *Uppsala University*, *Box 311*, *S-751 05 Uppsala*, *Sweden*
E-mail: svenolof@csd.uu.se

and

Bengt Jonsson

Department of Computer Systems, *Uppsala University*, *Box 325*, *S-751 05 Uppsala*, *Sweden*
E-mail: bengt@docs.uu.se

A compositional and fully abstract semantics for concurrent constraint programming is developed. It is the first fully abstract semantics which takes into account both non-determinism, infinite computations, and fairness. We present a simple concurrent constraint programming language, whose semantics is given by a set of reduction rules augmented with fairness requirements. In the fully abstract semantics we consider two aspects of a trace, viz. the function computed by the trace (the functionality) and the set of input and output data (the limit of the trace). We then derive the fully abstract semantics from the set of traces using a closure operation. We give two proofs of full abstraction; the first relies on the use of a syntactically infinite context. The second proof requires only a finite context, but assumes as input a representation of the function to be computed by the context. Finally, we examine the algebraic properties of the programming language with respect to the fully abstract semantics. It turns out that the non-deterministic selection operation can be defined using operations derived from parallel composition and the usual set-theoretic operations on sets of traces.

1. INTRODUCTION

A fully abstract semantics is a compositional semantics which identifies programs which have the same observable behaviour in any context. Whether a semantics is fully abstract or not depends on what is considered to be observable behaviour. We take a simple view on observability: *The observable behaviour of a process is its output*. As we consider infinite computations, the observable behaviour of a process may include an infinite collection of output generated during an infinite computation.

0890-5401/98 $25.00

The vehicle of our investigations is concurrent constraint programming [22, 32], which provides a simple and powerful model of asynchronous concurrent computation. The main feature of concurrent constraint programming (ccp) is that the store is seen as a constraint on the range of values that variables can assume, rather than as a particular mapping from variables to values. A *concurrent* constraint language adds primitives for communication between the components of a program. The communication primitives are now *ask* (check if a constraint is entailed by the store) and *tell* (add a new constraint to the store). A computation can be seen as an accumulation of (partial) information (constraints) to the store.

The basic communication paradigm in ccp is asynchronous. For several asynchronous concurrent computing paradigms, traces have become a standard basis for fully abstract semantics, e.g., for data-flow networks [17] or for general shared-variable programs [7]. De Boer *et al.* [9] give a general framework for the semantics of concurrent languages with asynchronous communication. They also note that the traditional failures model for CSP [6] is unnecessarily detailed when applied to programming languages which only allow asynchronous communication.

Intuitively, a trace of a program can be obtained from a computation by extracting the sequence of communication actions performed during the computation. In data-flow networks, a communication action is the reception or transmission of a data item on a channel; for shared-variable programs, a communication action is an atomic change to the global shared state. In ccp, it is natural to regard a communication action as the addition of information to the store.

In ccp, the set of traces of a program gives a complete description of the behaviour of the program in all possible contexts, but it contains too much detail; i.e., it is not fully abstract. We solve this problem by adding an operation that forms the downward closure of the set of traces with respect to a partial order. Intuitively, this partial order captures the notion that one trace contains less information than another. We then show that the semantics obtained by applying this closure operation to the trace semantics is compositional and fully abstract with respect to the result semantics, i.e., the observable behaviour as discussed above. A similar closure operation on traces has also been presented by Saraswat *et al.* [32], but that work only considers finite behaviour. In contrast our semantics handles infinite computations and the associated notion of fairness, and can be seen as a natural extension of [32] to the infinite case.

Earlier work on the semantics of concurrent constraint programming includes the work by de Boer and Palamidessi [10] who present a fully abstract semantics, but only for programs that exhibit only finite behaviours. They formulate the fully abstract semantics by means of structural operational semantics. de Boer *et al.* [11] have considered a variation of ccp with a restricted form of observability, the set of observable results is upward-closed, and given a fully abstract fixpoint semantics. However, for the general case, when arbitrary sets of observables are possible, they show that it is not possible to give a fully abstract fixpoint semantics. Jagadeesan *et al.* [16] consider a semantics of ccp with *angelic* nondeterminism, i.e., only successful branches of computations are considered, and give a fully abstract fixpoint semantics. In contrast, in this paper we allow for the possibility that a ccp process may interact with its environment. Since a failing computation may have

already performed communication we cannot ignore the behaviour of failed computations.

The remainder of the paper is organised as follows. Section 2 defines constraint systems. Section 3 gives the syntax of ccp together with some program examples. In Section 4 the operational semantics of ccp is defined. In Section 5 we give a formal definition of fairness. The observable behaviour of a ccp agent is defined in Section 6, where also a simple trace semantics is given. In Section 7 we review some results concerning closure operators and deterministic ccp programs. In Section 8 we define the abstract semantics and show that it is compositional and fully abstract. Section 9 gives an alternative proof of full abstraction. Section 10 examines the algebraic properties of ccp.

2. CONSTRAINTS

A constraint system consists of logical formulas, and rules for when a formula is entailed by a set of formulas, i.e., a store. We assume that a set of formulas in a store is represented as a conjunction of the formulas. Thus, the logical operations we have to reason about are conjunction, existential quantification (to deal with hiding), and implication (entailment).

As a basis for concurrent constraint programming, we need to find a mathematical structure which contains the desired formulas, and also satisfies the properties which are needed to apply the standard techniques of denotational semantics. It is necessary that the constraint system is *complete*, that is, for a chain of stronger and stronger constraints there should be a minimal constraint that is stronger than all constraints in the chain, and that the basic operations of the constraint system, existential quantification, conjunction, and implication, are continuous. Palmgren [28] gives a general method to construct a complete structure from an arbitrary structure so that the formulas valid in the constructed structure are exactly those that are valid in the original structure. However, here we use a simpler construction which directly gives a complete constraint system.

To get the appropriate constraint system we start with a set of formulas, closed under conjunction and existential quantification, and an interpretation that gives the truth values of formulas, given an assignment of values to (free) variables. Given this, we use ideal completion to derive the desired domain. The resulting structure satisfies all axioms of cylindric algebra [14] that do not involve negation.

In contrast, Saraswat *et al.* [32] choose an axiomatic approach, based on axioms from cylindric algebra and techniques from Scott's information systems [33] to specify the properties of a constraint system. The use of ideal completion to construct a constraint system that is closed under infinite limits has previously been employed by Carlson [8] and Kwiatkowska [20].

2.1. Mathematical Preliminaries

A *preorder* is a binary relation $\leqslant$ which is transitive and reflexive. Given a preorder $\leqslant$ over a set L, an *upper bound* of a set $X \subseteq L$ is an element $x \in L$ such that $y \leqslant x$ for all $y \in X$. The *least upper bound* of a set X, written $\bigvee X$, is an upper bound

x of X such that for any upper bound y of X, we have $x \leqslant y$. The concepts *lower bound* and *greatest lower bound* are defined dually. A function f over a preorder is *monotone* if $x \leqslant y$ implies $f(x) \leqslant f(y)$. For a preorder $(L, \leqslant)$ and $S \subseteq L$, let $S^u = \{x \mid y \in S, x \leqslant y\}$.

A *partial order* is preorder which is also antisymmetric. A *lattice* is a partial order $(L, \leqslant)$ such that every finite subset has a least upper bound and a greatest lower bound. A *complete lattice* is a partial order $(L, \leqslant)$ such that every subset has a least upper bound (this implies that every subset also has a greatest lower bound). A set $R \subseteq L$ is *directed* if every finite subset of R has an upper bound in R. A function f over a complete lattice L is *continuous* if for every directed set $R \subseteq L$ we have $\bigvee\{f(x) \mid x \in R\} = f(\bigvee R)$. For a complete lattice L, an element $x \in L$ is *finite* if for every directed set R such that $x \leqslant \bigvee R$, there is some $y \in R$ such that $x \leqslant y$. For a lattice L, let $\mathscr{K}(L)$ be the set of finite elements of L. A complete lattice L is *algebraic* if $x = \bigvee\{y \in \mathscr{K}(L) \mid y \leqslant x\}$ for all $x \in L$, i.e., all elements of L are either finite or the limit of a set of finite elements. Note that given an algebraic lattice $(L, \leqslant)$ and a monotone function f over $\mathscr{K}(L)$ we can easily extend f to a continuous function f' over L with $f'(x) = f(x)$, for $x \in \mathscr{K}(L)$, and $f'(x) = \bigvee\{f(y) \mid y \in \mathscr{K}(L), y \leqslant x\}$, for $x \in L \backslash \mathscr{K}(L)$.

2.2. Constraint Systems

DEFINITION 2.1. A *pre-constraint system* is a tuple $\langle F, Var, \models, C\rangle$, where F is a set (of *formulas*), Var is a set (of *variables*), C is a set (*the domain of values*), and $\models \subseteq Val \times F$ (a *valuation*), where Val is the set of functions from Var to C (*assignments*).

If X and Y are variables and ϕ and ψ are members of F the following formulas should also be members of F.

$$X = Y \qquad \exists_X \phi \qquad \phi \wedge \psi$$

Given an assignment V, formulas ϕ and ψ, and variables X and Y, we expect the valuation $\models$ to satisfy the following.

1. $V \models X = Y$ iff $V(X) = V(Y)$.
2. $V \models \exists_X \phi$ iff $V' \models \phi$ for some assignment V' such that $V(X') = V(X')$ whenever $X \neq X'$.
3. $V \models \phi \wedge \psi$ if $V \models \phi$ and $V \models \psi$.

Note that the valuation for a formula $\phi \wedge \psi$ is uniquely determined by the valuations for the formulas ϕ and ψ. Similarly, given the valuation for ϕ we can determine the valuation for $\exists_X \phi$. Thus, we do not need to specify the valuations for conjunctions and existential quantifications in the definition of a pre-constraint system.

EXAMPLE 2.2. Let C be the set of natural numbers. Let the set of formulas be the smallest set that satisfies the axioms and contains the formula $X = n$, for each variable X and $n \in C$. Say that $V \models X = n$ iff $V(X) = n$.

The general definition allows very powerful constraint systems, where the basic operations can be computationally expensive, or even uncomputable. If we want a concurrent language that can be implemented efficiently, we should of course choose a constraint system where the basic operations (adding a constraint to the store and entailment) have efficient implementations. The following construction, sometimes referred to as the "term model," gives us a ccp language with power and expressiveness comparable to concurrent logic languages such as GHC and Parlog.

EXAMPLE 2.3. Suppose we have a set of constant symbols $\{a, ...\}$ and function symbols $\{f, ...\}$. Define a set of *expressions* according to

1. a is an expression, for any constant symbol a.
2. $f(E_1, ..., E_n)$ is an expression, if f is an n-ary-function symbol and $E_1, ..., E_n$ are expressions.
3. X is an expression, if X is a variable.

The formulas in F are simply formulas of the form $E_1 = E_2$, where E_1, E_2 are expressions. Let C be the set of expressions that do not contain variables.

To judge whether $V \models E_1 = E_2$ holds for an assignment V and expressions E_1 and E_2, simply replace each variable X in E_1 and E_2 with the corresponding value given by $V(X)$. Then $V \models E_1 = E_2$ holds if and only if the resulting expressions are syntactically equal.

We define a preorder $\preccurlyeq$ between formulas by $\phi \preccurlyeq \psi$ iff for any $V \in Val$ such that $V \models \psi$, we have $V \models \phi$. One can think of $\phi \preccurlyeq \psi$ as meaning that ϕ is weaker than ψ, or that ψ implies ϕ. This gives immediately an equivalence relation $\phi \equiv \psi$ defined by $\phi \preccurlyeq \psi$ and $\psi \preccurlyeq \phi$. Next we transform the preorder of formulas into a domain where equivalent formulas are identified and elements are added so each increasing chain has a limit.

DEFINITION 2.4. A *constraint* is a non-empty set c of formulas, such that

1. if $\phi \in c$, and $\psi \preccurlyeq \phi$, then $\psi \in c$, and
2. if $\phi, \psi \in c$, then $\phi \wedge \psi \in c$.

For a formula ϕ let $[\phi] = \{\psi \mid \psi \preccurlyeq \phi\}$. Clearly $[\phi]$ is a constraint. If we have a directed set R of constraints, then it follows from the definition of constraints that $\bigcup R$ is also a constraint.

The set of constraints form an algebraic lattice under the $\subseteq$ ordering, with least element $\bot$ being the set of all formulas which hold under all assignments, that is, $\{X = X, Y = Y, ...\}$. We use the usual relation symbol $\sqsubseteq$ for inclusion between constraints, so that $c \sqsubseteq d$ if and only if $c \subseteq d$, and the symbol $\bigsqcup$ for least upper bound. Say that a constraint c is *finite* if whenever R is a directed set such that $c \sqsubseteq \bigsqcup R$, there is some $d \in R$ such that $c \sqsubseteq d$. Let $\mathcal{U}$ be the set of constraints, and $\mathcal{K}(\mathcal{U})$ the set of finite constraints. Note that for formulas ϕ and ψ, we have $[\phi] \sqcup [\psi] = [\phi \wedge \psi]$. We can thus see least upper bound as an extension of conjunction. The finite constraints are exactly those constraints that can be given in the form $[\phi]$, for some formula ϕ. Also note that each constraint is either finite, or a limit of a directed set of finite constraints, which implies that the constraints form

an algebraic lattice. Existential quantification is extended into a continuous function over the constraints according trio the following rules.

1. $\exists_X([\phi]) = [\exists_X \phi]$, for formulas ϕ.
2. $\exists_X(\bigsqcup R) = \bigsqcup_{d \in R} \exists_X(d)$, for directed sets $R \subseteq \mathscr{K}(\mathscr{U})$.

2.3. Examples of Constraint Systems

EXAMPLE 2.5. Consider the term model mentioned in a previous example. The ideal completion gives us a new structure that is quite similar to the one we had previously, except that we now can find a constraint c such that c holds if and only if all of the formulas

$$\exists_Y(X = f(Y)), \qquad \exists_Y(X = f(f(Y))), \qquad \exists_Y(X = f(f(f(Y)))), \ldots$$

hold.

EXAMPLE 2.6 (Rational intervals). Let the domain of values be the rational numbers, the formulas be of the form $X \in [r_1, r_2]$, where r_1 and r_2 are rational numbers. Let the valuation $\models$ be such that $V \models X \in [r_1, r_2]$ iff $r_1 \leqslant V(X) \leqslant r_2$. It should be clear that in this constraint system, entailment is computable.

2.4. Properties of the Constraint System

We give some algebraic properties of constraint systems, corresponding largely to the axioms of cylindric algebra [14].

PROPOSITION 2.7. *Given a pre-constraint system* $\langle F, Var, \models, C \rangle$, *let the lattice* $\langle \mathscr{U}, \sqsubseteq \rangle$ *be the corresponding domain of constraints, with* $\bot$ *and* $\top$ *the least and greatest elements of* $\mathscr{U}$. *The following postulates are satisfied for any constraints* $c, d \in \mathscr{U}$ *and any variables* $X, Y, Z \in Var$.

1. *the structure* $(\mathscr{U}, \sqsubseteq)$ *forms an algebraic lattice.*
2. $\exists_X$ *is a continuous function* $\exists_X: \mathscr{U} \to \mathscr{U}$,
3. $\exists_X(\top) = \top$,
4. $\exists_X(c) \sqsubseteq c$,
5. $\exists_X(c \sqcup \exists_X(d)) = \exists_X(c) \sqcup \exists_X(d)$,
6. $\exists_X(\exists_Y(c)) = \exists_Y(\exists_X(c))$,
7. $(X = X) = \bot$,
8. $(X = Y) = \exists_Z(X = Z \sqcup Z = Y)$, *for* Z *distinct from* X *and* Y,
9. $c \sqsubseteq (X = Y) \sqcup \exists_X(X = Y \sqcup c)$, *for* X *and* Y *distinct.*

Items 3–9 are borrowed from cylindric algebra. However, the structure is not necessarily a cylindric algebra, since a cylindric algebra is required to satisfy the axioms of Boolean algebra and must thus be a distributive lattice, while it is possible to construct a constraint system which is not distributive. For example, the

constraint system derived from the pre-constraint system in Example 2.2 is not a distributive lattice, since it contains the sub-lattice $\{\mathbf{true}, X=1, X=2, X=3, \mathbf{false}\}$.

Remark. We have given a general framework for the construction of constraint systems. In the proofs, we will refer to the following properties of a constraint system: that it satisfies the axioms of cylindric algebra listed above, and that it forms an algebraic lattice. In contrast, Saraswat *et al.* [32] require in their semantics for non-deterministic ccp that the constraint system should be *finitary*; i.e., that for each finite constraint there should only be a finite set of smaller finite constraints. As an example of a constraint system that is *not* finitary, they mention the constraint system of rational intervals, as described in Example 2.6.

3. SYNTAX OF CCP

We assume a set $\mathcal{N}$ of procedure symbols $p, q, \ldots$. The syntax of an *agent* A is given as follows, where c ranges over finite constraints, and X over variables.

$$
\begin{aligned}
A ::= {} & c \quad | \\
& \bigwedge_{j \in I} A^j \quad | \\
& (c_1 \Rightarrow A_1 \,\square \cdots \square\, c_n \Rightarrow A_n) \quad | \\
& \exists_X^c A \quad | \\
& p(X)
\end{aligned}
$$

A tell constraint, written c, is assumed to be a member of $\mathcal{K}(\mathcal{U})$. The conjunction

$$\bigwedge_{j \in I} A^j$$

of agents, where I is assumed to be countable, represents a parallel composition of the agents A^j. We will use $A^1 \wedge A^2$ as a shorthand for $\bigwedge_{j \in \{1,2\}} A^j$. We use superscripts to avoid confusion with subscripts representing positions in a computation (see Section 4). An agent $(c_1 \Rightarrow A_1 \,\square \cdots \square\, c_n \Rightarrow A_n)$ represents a selection. If one of the ask constraints c_i becomes true, the corresponding agent A_i may be executed. Agents of the form $\exists_X^c A$ represent agents with local data. The variable X is local, which means that the value of X is not visible to the outside. The constraint c is used to represent the constraint on the local value of X between computation steps.

Note that the syntax for agents describes both agents appearing in a program, and agents appearing as intermediate states in a computation. However, we will assume that agents of the form $\exists_X^c A$ occurring in a program or in the initial state of a computation will always have $c = \bot$. When this is the case, the local store can be omitted and the agent written $\exists_X A$.

A *program* Π is a set of definitions of the form $p(X) :: A$, where each procedure symbol p occurs in the left-hand side of exactly one definition in the program.

Remark. To simplify the presentation, we only consider definitions with one argument. If we assume a suitable constraint system, such as the term model, we can use a function symbol (e.g., f) as a tuple constructor and let the formula $f(E_1, ..., E_n)$ represent a tuple of the n arguments.

3.1. Examples

We give some examples of concurrent constraint programs to help the reader get an intuitive understanding of ccp. The programs assume the term model, defined in Example 2.5.

We say that a variable X is *bound to* a value v if the current store c is such that for any assignment V, we have $V \models c$ iff $V(X) = v$. Similarly, we say that we *bind* a variable V to a value v if we by adding constraints to the store make sure that V is bound to v in the resulting store.

EXAMPLE 3.1 (Non-determinism). As an example of a procedure that is *not* deterministic, consider

$$\text{erratic}(X) :: (\textbf{true} \Rightarrow X = 0 \ \square \ \textbf{true} \Rightarrow X = 1).$$

(Here we use **true** as a shorthand for some arbitrary constraint that always holds, like $X = X$.) The agent erratic(X) will either bind the variable X to 0 or to 1. It is important to keep in mind that the language does not define the procedure to satisfy any probabilistic properties or fairness conditions in selections; for example, an implementation where the agent always binds X to 0 is correct.

EXAMPLE 3.2 (McCarthy's ambiguity operator). As mentioned in the previous section, our language only allows procedures with one argument. It is easy to overcome this limitation using dedicated function symbols to represent groups of two or more arguments. We will assume that this is applied as a "syntactic sugar."

Assuming this, we can give a more interesting example of a non-deterministic procedure definition.

$$\text{amb}(X, Y, Z) :: (\text{number}(X) \Rightarrow Z = X \ \square \ \text{number}(Y) \Rightarrow Z = Y)$$

The amb procedure, which is inspired by McCarthy's ambiguity operator [23], waits until either the first or the second argument is instantiated to a number, and then sets the third argument equal to the one of the two first that was defined. If both the first and the second arguments are numbers, the choice is arbitrary. (We assume that there is a predicate "number" which holds for numbers and nothing else.) Consider the agent

$$X = 5 \wedge \text{amb}(X, Y, Z).$$

When the agent is run, the final store will be

$$X = 5 \wedge Z = 5.$$

Of course, if the agent above is put into a context where Y is bound to a number, it is possible that the final store has Z bound to Y.

In the subsequent examples, we will use the same notation for lists as Prolog. [ ] is the empty list, $[X \mid Y]$ is a list where the first element is X and the rest of the list is Y. A list of n elements can be written $[X_1, X_2, ..., X_n]$, where X_1 is the first element and so on. So, if $Y = [2, 3]$ and $X = 1$ it follows that $[X \mid Y] = [1 \mid [2, 3]] = [1, 2, 3]$.

EXAMPLE 3.3 (The "ones" program). As an example of a simple program that produces an infinite result, consider the following.

$$\text{ones}(X) :: \exists_Y (X = [1 \mid Y] \wedge \text{ones}(Y))$$

Running an agent ones(X) should generate longer and longer approximations of an infinite list of ones, i.e., stores where constraints of the form

$$\exists_Y X = [\underbrace{1, 1, ..., 1}_{n \text{ times}} \mid Y]$$

are entailed, for increasingly larger n. The "final" result is then the limit of these stores, i.e., the store in which the constraint

$$X = [1, 1, 1, ...]$$

is entailed.

EXAMPLE 3.4 (Two-way communication: The "lazy-ones" program). The examples we have shown could have been written in most asynchronous concurrent programming languages. However, concurrent constraint languages and concurrent logic languages allow a kind of two-way communication, which increases the expressiveness.

The program in this example also generates a list of ones, however, only when requested.

In this program, the matching of input arguments is performed by ask constraints which contain existentially quantified variables. In general, to inquire whether X is of the form $f(Y)$, for some Y, we use the ask constraint $\exists_Y (X = f(Y))$. In the program we use the ask constraint $\exists_A \exists_{X_1} (X = [A \mid X_1])$ to check whether X variable is bound to a list of at least one element. (In this program, and in other program examples, we follow the convention that capital letters are variables.)

$$\begin{aligned}
&\text{lazy_ones}(X) :: \\
&\quad (\exists_A \exists_{X_1} (X = [A \mid X_1]) \\
&\qquad \Rightarrow \exists_{X_1} (X = [1 \mid X_1] \wedge \text{lazy_ones}(X_1)) \\
&\quad \square\ X = [\,] \\
&\qquad \Rightarrow \textbf{true})
\end{aligned}$$

The first condition in the selection tests if X is bound to a list of at least one element. If this is the case, and the corresponding branch is taken, a constraint saying that the first element of X is a one will be added to the store. Then "lazy_ones" is called recursively with the rest of the list as argument. The second condition is for the case when X is bound to the empty list. The corresponding branch is the empty constraint **true**; i.e., when $X = [\ ]$ the agent lazy ones(X) will terminate.

Now, consider an execution of the call lazy_ones(X). Suppose that X is unbound, i.e., that there is no information about X in the store. Neither of the two conditions hold, so the call cannot execute. Suppose now that X is bound to the list $[A_1, A_2 \mid X_1]$. Now it holds that X is a list with at least one element and thus the first alternative may be selected. Then A_1 is bound to 1, lazy_ones is called recursively with $[A_2 \mid X_1]$ as argument, A_2 is bound to 1 and then the agent sits down and waits for X_1 to be found.

The lazy_ones program is of course not a very interesting program in itself, but the technique of using a partially instantiated structure to allow a form of two-way communication between processes has many possible applications. Shapiro [35] describes how it is possible to write concurrent logic programs in an "object-oriented" style, where an object is represented as a process which has a local state and reads and responds to a stream of messages. Examples of more substantial clp programs can be found in the textbooks on clp, for example reference [12]. We would again like to point out that the differences between clp and ccp are mostly syntactical. Ccp with the term model as constraint system is very close to concurrent logic programming languages such as GHC and Strand.

EXAMPLE 3.5 (Unbounded non-determinism). This last example is inspired by Park [29], who showed that in a language with non-determinism and some notion of fairness it is possible to write a program that exhibits unbounded non-determinism. We assume that the constraint system contains the natural numbers in the domain of values, and that constraints of the forms $X = 0$ and $X = Y + 1$ are allowed.

$$
\begin{aligned}
p(X) :: \exists_A A = 0 & \\
& \wedge\ p_1(A, X) \\
p_1(A, X) :: \exists_A \exists_{X_1} A_1 = A + 1 & \\
& \wedge\ p_1(A_1, X_1) \\
& \wedge\ \mathrm{amb}(A, X_1, X).
\end{aligned}
$$

In the successive recursive call to p_1, the first argument will always be bound to an integer. Thus, the call to amb is guaranteed to terminate, and each recursive call to p_1 will bind its second argument to an integer. Clearly one possible result of a call of the form $p_1(n, X)$, where n is bound to an integer is to bind X to n. However, noting that a call $p_1(n, X)$ will result in the execution of $p_1(n+1, X_1) \wedge \mathrm{amb}(n, X_1, X)$ we see that it is possible for the recursive call to bind X_1 to $n+1$.

Thus, the call to amb may bind X to $n+1$. It follows by an inductive argument that a call $p_1(n, X)$ may bind X to any integer greater or equal to n. Thus a call $p(X)$ will always bind X to an integer, and may bind X to any integer greater or equal to zero.

The program given above may compute for an arbitrarily long time but will always produce a result. It is of course possible to write a similar program that will either produce an arbitrary integer or compute for ever but fail to produce a result. To distinguish the behaviour of that program from the one given above we must consider observations of infinite computations.

4. OPERATIONAL SEMANTICS

A *configuration* is a pair $A : c$ consisting of an agent A acting on a finite constraint c. The latter will be referred to as the *store* of the configuration. The operational semantics is given through a relation $\rightarrow$ over configurations, assuming a program Π. For any computation step $A : c \rightarrow A' : c'$, the constraint c' will always contain more information than c, i.e., $c \sqsubseteq c'$, so a computation step is never destructive.

We define $\rightarrow$ in the usual style of structural operational semantics [30].

1. The *tell constraint* simply adds new information (itself) to the store,

$$c : d \rightarrow c : c \sqcup d.$$

2. A *conjunction* of agents is executed by interleaving the execution of its components,

$$\frac{A^k : c \rightarrow B^k : d, \qquad k \in I}{\bigwedge_{j \in I} A^j : c \rightarrow \bigwedge_{j \in I} B^j : d},$$

where $B^j = A^j$, for $j \in I \backslash \{k\}$.

3. If one of the ask constraints in a *selection* is entailed by the current store, the selection can be reduced to the corresponding agent,

$$\frac{c_i \sqsubseteq c}{(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n) : c \rightarrow A_i : c}.$$

4. A configuration with an existentially quantified agent $\exists_X^c A : d$ is executed one step by doing the following. Apply the function $\exists_X$ to the present store d, hiding any information related to the variable X, combine the result $\exists_X(d)$ with the local data (given by c), to obtain a local store. A computation step is performed in the local store, which gives a new local store (c' say). To transmit any results to the global store, the function $\exists_X$ is again applied to hide any information relating to

the variable X. The constraint thus obtained is combined with the previous global store d. The local store c' is stored as part of the existential quantification

$$\frac{A : c \sqcup \exists_X(d) \to A' : c'}{\exists_X^c A : d \to \exists_X^{c'} A' : d \sqcup \exists_X(c')}.$$

5. A *call* to a procedure is reduced to the body of its definition,

$$p(X) : c \to A[X/Y] : c,$$

where the definition $p(Y) :: A$ is a member of Π, and the "substitution" $A[X/Y]$ is a shorthand[1] for

$$\exists_\alpha(\alpha = X \wedge \exists_Y(\alpha = Y \wedge A)),$$

where α is a variable that does not occur in the program.

4.1. Some Simple Computation Examples

We apply the computation rules to some simple agents.

First, a conjunction of a selection and a tell constraint. Consider the configuration

$$(X = 5 \Rightarrow Y = 7) \wedge X = 5 : \bot.$$

The selection cannot execute, since the ask constraint is not entailed by the store (that is, $\bot$). The tell constraint is executable, so we can perform the computation step

$$\begin{aligned}(X = 5 \Rightarrow Y = 7) \wedge X = 5 : \bot \\ \to (X = 5 \Rightarrow Y = 7) \wedge X = 5 : X = 5.\end{aligned}$$

(The tell constraint still remains in the conjunction, even though it is now redundant.) Now as the constraint $X = 5$ has been added to the store, the selection can execute, as the ask constraint of its only alternative is entailed.

$$(X = 5 \Rightarrow Y = 7) \wedge X = 5 : X = 5 \to Y = 7 \wedge X = 5 : X = 5$$

As we have replaced the selection by its only branch, we see immediately that another computation step is possible,

$$Y = 7 \wedge X = 5 : X = 5 \to Y = 7 \wedge X = 5 : X = 5 \wedge Y = 7.$$

Note that the constraint $X = 5 \wedge Y = 7$ is equivalent to $(X = 5) \sqcup (Y = 7)$. We chose the former notation since it is more readable.

[1] We could use $A[X/Y] \equiv \exists_Y(Y = X \wedge A)$, if we knew that the variables X and Y were always distinct.

Next we consider a computation which involves hidden data. The reader may find it helpful to take a look at the computation rules for existential quantifications before proceeding. Let A be the agent

$$(X=5 \Rightarrow Y=7) \wedge (Y=7 \Rightarrow Z=3),$$

and consider the configuration

$$\exists_Y A : \bot.$$

(The local data of the existential quantification is $\bot$.) To perform a computation step by the existential quantification, we must check if the configuration

$$A : \bot$$

can perform a computation step. Clearly, since the agent A consists of a conjunction of two selections, and neither of the tests (ask constraints) are entailed by the store, it follows that A can not perform any computation step. Suppose now that input arrives from the outside, and we find that the store contains the constraint $X=5$. To perform a computation step with the configuration

$$\exists_Y A : X=5,$$

we consider the "local" configuration

$$A : X=5,$$

where the store was obtained by $\bot \sqcup \exists_Y (X=5) = (X=5)$. We see that the test of the first selection of A is entailed by the store, so we can perform the computation step

$$A : X=5 \rightarrow Y=7 \wedge (Y=7 \Rightarrow Z=3) : X=5.$$

Thus, we have the computation step

$$\exists_Y A : X=5 \rightarrow \exists_Y^{X=5} (Y=7 \wedge (Y=7 \Rightarrow Z=3)) : X=5.$$

To see if the existential quantification can do another step, we look again at the local configuration

$$Y=7 \wedge (Y=7 \Rightarrow Z=3) : X=5.$$

We can perform a local step

$$\begin{aligned} &Y=7 \wedge (Y=7 \Rightarrow Z=3) : X=5 \\ &\quad \rightarrow Y=7 \wedge (Y=7 \Rightarrow Z=3) : X=5 \wedge Y=7, \end{aligned}$$

which corresponds to the step

$$\exists_Y^{X=5}(Y=7 \wedge (Y=7 \Rightarrow Z=3)) : X=5$$
$$\rightarrow \exists_Y^{X=5\,Y=7\,Y=7}(Y=7 \wedge (Y=7 \Rightarrow Z=3)) : X=5$$

at the higher level. Since $\exists_Y(X=5 \wedge Y=7)$ is equal to $(X=5)$, the constraint concerning the value of Y is not visible outside the quantification. However, the local value of Y is recorded in the local store. Last two steps are straightforward. We do a local step

$$Y=7 \wedge (Y=7 \Rightarrow Z=3) : X=5$$
$$\rightarrow Y=7 \wedge Z=3 : X=5 \wedge Y=7,$$

corresponding to the global step

$$\exists_Y^{X=5 \wedge Y=7}(Y=7 \wedge (Y=7 \Rightarrow Z=3)) : X=5$$
$$\rightarrow \exists_Y^{X=5 \wedge Y=7}(Y=7 \wedge Z=3) : X=5,$$

and finally the local step

$$Y=7 \wedge Z=3 : X=5 \wedge Y=7$$
$$\rightarrow Y=7 \wedge Z=3 : X=5 \wedge Y=7 \wedge Z=3,$$

which corresponds to the global step

$$\exists_Y^{X=5 \wedge Y=7}(Y=7 \wedge Z=3) : X=5$$
$$\rightarrow \exists_Y^{X=5\,Y=7\,Y=3}(Y=7 \wedge Z=3) : X=5 \wedge Z=3.$$

Thus, we finally reach a configuration in which the store is $X=5 \wedge Z=3$.

4.2. Computations

Using the operational definition we can specify the set of computations. The basic idea is that a computation is the result of applying a sequence of interleaved computation steps and input steps to a configuration.

DEFINITION 4.1. Assuming a program Π, a *computation* is an infinite sequence of configurations $(A_i : c_i)_{i \in \omega}$ such that for all $i \geqslant 0$, we have either $A_i : c_i \rightarrow A_{i+1} : c_{i+1}$ (*a computation step*), or $A_i = A_{i+1}$ and $c_i \sqsubseteq c_{i+1}$ (*an input step*). An input step from $A : c$ to $A : c'$ such that $c = c'$ is an *empty input step*. A computation where all input steps are empty is a *non-interactive computation*.

Note that some steps are both computation steps and input steps. For example, going from

$$X=5 : \bot \qquad \text{to} \qquad X=5 : X=5$$

can be done either in a computation step, or in an input step.

In the following text, we will, leave out references to the program Π when we can do so without causing ambiguities.

Remark. According to the definition above, all computations are infinite. However, since one can see a finite computation as an infinite computation which ends in an infinite sequence of empty input steps, we do not lose in generality by only considering infinite computations.

5. FAIRNESS

The structured operational semantics does not in itself define fairness. It is necessary to use some device to restrict the set of computations, thus avoiding, for example, situations where one agent in a conjunction is able to perform a computation step but is never allowed to do so.

Intuitively, a computation is fair if every agent that occurs in it and is able to perform some computation step will eventually perform some computation step. However, this intuitive notion is difficult to formalize directly. What does it mean that an agent is able to perform a computation step? Computation steps are performed on configurations, not on agents. Also, this requirement should not apply to alternatives in a selection, since an agent occurring in an alternative should not be executed until (and if) that alternative is selected. Third, what happens if one has a computation where an agent A occurs in many positions in every configuration in the computation? A direct formalization of the intuitive fairness requirement would fail to differentiate between different occurrences of the same agent, so a computation might incorrectly be considered fair if it performed computation steps on some occurrences of the agent A and ignored other occurrences of A.

How should we specify the set of fair computations? First, note that a computation can often be considered to contain other computations. For example, to perform a computation step with a process $A \wedge B : c$, it is necessary to perform computation steps with either of the processes $A : c$ and $B : c$. The view of a computation as a composition of computations leads us to the following definitions.

DEFINITION 5.1. Let the relation *immediate inner computation of* be the weakest relation over ω-sequences of configurations which satisfies the following.

1. $(A_i^k : c_i)_{i \in \omega}$ is an immediate inner computation of $(\bigwedge_{j \in I} A_i^j : c_i)_{i \in \omega}$, for $k \in I$.

2. $(A_i : c_i \sqcup \exists_X(d_i))_{i \in \omega}$ is an immediate computation of the computation $(\exists_X^{c_i} A_i : d_i)_{i \in \omega}$.

The relation *inner computation of* is defined to be the transitive and reflexive closure over the relation "immediate inner computation of."

We would expect the inner computations of a computation to also be computations.

PROPOSITION 5.2. *If $(A_i : c_i)_{i \in \omega}$ is computation, and $(B_i : d_i)_{i \in \omega}$ is an inner computation of $(A_i : c_i)_{i \in \omega}$, then $(B_i : d_i)_{i \in \omega}$ is also a computation.*

The proposition follows from the operational semantics.

We will define the fairness requirement in a bottom-up fashion by giving a sequence of auxiliary definitions which capture different aspects of fairness. First, the weakest and simplest fairness property, top-level fairness. A computation is top-level fair unless the first agent of the computation is a tell constraint that is never added to the store, or a selection which has an alternative that can be selected, but no alternative is selected, or a call that is never reduced to its definition. Using top-level fairness we can specify initial fairness, which concerns agents occurring as a part of the first agent, and finally the actual definition of fairness.

DEFINITION 5.3. A computation $(A_i : c_i)_{i \in \omega}$ is *top-level fair* when the following holds.

1. If $A_0 = p(X)$, there is an $i \geqslant 0$ such that $A_i \neq A_0$.
2. If $A_0 = c$, there is an $i \geqslant 0$ such that $c_i \sqsupseteq c$.
3. If $A_0 = (d_1 \Rightarrow B_1 \,\square\, \cdots \,\square\, d_n \Rightarrow B_n)$, and $d_j \sqsubseteq c_0$ for some $j \leqslant n$, then there is an $i \geqslant 0$ such that $A_i \neq A_0$.

A computation is *initially fair* if all its inner computations are top-level fair. A computation is *fair* if all its proper suffixes are initially fair.

PROPOSITION 5.4. *It is easy to verify that the fairness requirement satisfies the following properties.*

1. *If one suffix of a computation is top-level fair then the computation is top-level fair.*
2. *If one suffix of a computation is initially fair, then the computation is initially fair.*
3. *If one suffix of a computation is fair, then the computation is fair.*
4. *All inner computations of a fair computation are fair.*
5. *A computation whose immediate inner computations are fair, and all suffixes are top-level fair, is fair.*

(The second condition of Item 5 cannot be omitted; some computations do not have immediate inner computations.)

Item 1 follows from the definition of fairness and the operational semantics. Item 2 follows from the definition of inner computations and Item 1. Item 3 follows from the definition of fairness and Item 2. Items 4 and 5 follow from the definition of fairness.

6. RESULT AND TRACE SEMANTICS

We present two semantics based on the operational model of ccp.

First, the *result semantics*, which considers only the relation between the initial and final constraint stores in a computation. The result semantics is intended to give the *observable behaviour* of an agent. The result semantics is of course not compositional, since it does not capture interaction between agents.

The second semantics is the *trace semantics*, where a process is represented by a set of traces. Each trace is an infinite sequence of stores together with information on which steps in the computation are computation steps and which are input steps. Since the trace semantics records interaction between processes one would expect the trace semantics to be compositional, and this is indeed the case.

6.1. Result Semantics

Consider the situation where we run an agent without interaction with other agents. If the agent terminates, the result of the computation is the final contents of the store. If the agent does not terminate, we record the limit of the successive stores of the computation and say that the limit is the result of the computation.

The result semantics is given by a function $\mathscr{R}_{\Pi}$: AGENT $\to \mathscr{K}(\mathscr{U}) \to \mathscr{P}(\mathscr{U})$ which gives the set of all possible results that can be computed given a program Π, an agent A, and an initial store c,

$$\mathscr{R}_{\Pi}[\![A]\!]\, c = \left\{ \bigsqcup_{i \in \omega} c_i \mid (A_i : c_i)_{i \in \omega} \text{ is a fair non-interactive computation with } A_0 : c_0 = A : c. \right\}$$

It can be seen that for an infinite computation, we define the final constraint store as the limit of the intermediate constraint stores that occur during a computation. We find this very reasonable for a constraint programming language, since an arbitrary finite approximation of the "final" constraint store can be obtained by waiting long enough for the computation to proceed. This property does not hold for shared-variable programs in general where the information in the store does not have to be monotonously increasing.

6.2. Traces

A computation is defined to be a sequence of configurations $(A_i : c_i)_{i \in \omega}$, where the stores (the c_i's) are the only part of the computation visible to the outside. A computation can go from $A_i : c_i$ to $A_{i+1} : c_{i+1}$ either by performing a computation step, or by receiving input, and this distinction is of course relevant when comparing behaviours of agents. In the trace semantics, an agent is represented by a set of traces, where each trace is an infinite sequence of stores together with information on which steps in the computation are computation steps and which are input steps.

DEFINITION 6.1. A *trace* t is a pair $t = (v(t), r(t))$, where $v(t)$ is an ω-chain in $\mathscr{K}(\mathscr{U})$ and $r(t) \subseteq \omega$. The set of traces is denoted TRACE.

The *trace of a computation* $(A_i : c_i)_{i \in \omega}$ is a trace $t = ((c_i)_{i \in \omega}, r)$, where the step from $A_i : c_i$ to $A_{i+1} : c_{i+1}$ is a computation step when $i \in r$, and an input step when $i \notin r$. We will sometimes use the notation $v(t)_i$ to refer to the ith element of the store sequence of t.

The trace semantics of an agent A, assuming a program Π, is defined as follows.

DEFINITION 6.2. The function $\mathcal{O}_\Pi : \text{AGENT} \to \mathcal{P}(\text{TRACE})$ is defined so that $t \in \mathcal{O}_\Pi[\![A]\!]$, iff t is the trace of a fair computation $(A_i : c_i)_{i \in \omega}$, where $A_0 = A$.

When the above holds, we say that the computation $(A_i : c_i)_{i \in \omega}$ *connects* the trace t to the agent.

6.2.1. *Operational semantics of simple agents.* The operational semantics of tell constraints and calls can be given directly.

PROPOSITION 6.3. *For a tell constraint c, we have $t \in \mathcal{O}_\Pi[\![c]\!]$ iff*

1. $v(t)_i \sqsupseteq c$, *for some* $i \in \omega$ *and*
2. $v(t)_{i+1} = v(t)_i \sqcup c$, *for all* $i \in r(t)$.

The proposition follows from the fact that in each computation step $c : d_i \to c : d_{i+1}$, we have $d_{i+1} = d_i \sqcup c$. Fairness guarantees that the limit of the trace will be stronger than c.

PROPOSITION 6.4. *For a call $p(X)$, we have $\mathcal{O}_\Pi[\![p(X)]\!] = \mathcal{O}_\Pi[\![A[X/Y]]\!]$ where the definition of p in the program Π is $p(Y) :: A$.*

The proposition follows from the fact that the only computation step for a call is the reduction of the call to the corresponding procedure body, and from the fact that fairness requires that this step must be taken.

6.3. Compositionality

The trace semantics allows a compositional definition, as expressed in the following propositions. The proofs are based on the computation rules and the fairness requirements.

PROPOSITION 6.5. *Assume an agent $\bigwedge_{j \in I} A^j$. For a trace t we have $t \in \mathcal{O}_\Pi[\![\bigwedge_{j \in I} A^j]\!]$ iff there are $t_j \in \mathcal{O}_\Pi[\![A^j]\!]$ for $j \in I$, such that $v(t_j) = v(t)$ for $j \in I$, $r(t) = \bigcup_{j \in I} r(t_j)$, and $r(t_i) \cap r(t_j) = \varnothing$, for $i, j \in I$ such that $i \neq j$.*

Proof. ($\Rightarrow$) We know that there is a fair computation $(\bigwedge_{j \in I} A_i^j : c_i)_{i \in \omega}$ that connects t to A. By the operational semantics, we have a family of inner computations $\{(A_i^j : c_i)_{i \in \omega}\}_{j \in I}$. For each $i \in r(t)$ there is a $k_i \in I$ such that $A_i^{k_i} : c_i \to A_{i+1}^{k_i} : c_{i+1}$ is a computation step, and for $j \in I \backslash \{k_i\}$, there is an input step from $A_i^j : c_i$ to $A_{i+1}^j : c_{i+1}$. For $i \in \omega \backslash r(t)$ it is easy to see that the step from $A_i^j : c_i$ to $A_{i+1}^j : c_{i+1}$ is an input step for all $j \in I$.

For all $j \in I$, each computation $(A_i^j : c_i)_{i \in \omega}$ is fair, since all inner computations of a fair computation are fair (Proposition 5.4). For each $j \in I$, let the trace t_j be such that $v(t_j) = v(t)$, and $r(t_j) = \{i \in r(t) \mid k_i = j\}$. It is easy to check that the family of traces $\{t_j\}_{j \in I}$ satisfies the right-hand side of the proposition.

($\Leftarrow$) We know that for each $j \in I$ there is a fair computation $(A_i^j : c_i)_{i \in \omega}$ that connects t_j to A_j. For each $i \in r(t)$ there is a $k_i \in I$ such that $i \in r(t_{k_i})$ but $i \notin r(t_j)$, for $j \in I \backslash \{k_i\}$. By the computation rules it follows that

$$\bigwedge_{j \in I} A_i^j : c_i \to \bigwedge_{j \in I} A_{i+1}^j : c_{i+1},$$

for all $i \in r(t)$. In the case that $i \in \omega \backslash r(t)$, it follows that $i \in \omega \backslash r(t_j)$, for all $j \in I$, and thus all computations $(A_i^j : c_i)_{i \in \omega}$ perform input steps at position i, which implies that $\bigwedge_{j \in I} A^j = \bigwedge_{j \in I} A_{i+1}^j$, from which follows that we can construct a computation $(\bigwedge_{j \in I} A_i^j : c_i)_{i \in \omega}$. Fairness follows from the fact that all immediate inner computations of the constructed computation are fair. ▮

PROPOSITION 6.6. *Suppose we have an agent* $\exists_X A$. *For any trace* t, *we have* $t \in \mathcal{O}_\Pi[\![\exists_X A]\!]$ *iff there is a trace* $u \in \mathcal{O}_\Pi[\![A]\!]$ *such that with* $v(t) = (d_i)_{i \in \omega}$ *and* $v(u) = (e_i)_{i \in \omega}$, *we have*

1. $r(t) = r(u)$
2. $e_0 = \exists_X(d_0)$,
3. *for* $i \in r(t)$, $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, *and*
4. *for* $i \in \omega \backslash r(t)$, $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$.

Proof. ($\Rightarrow$) Suppose $t \in \mathcal{O}_\Pi[\![\exists_X A]\!]$. There is a computation $(\exists_X^{c_i} : d_i)_{i \in \omega}$ that connects the trace t to the agent A. The computation has an inner computation $(A_i : c_i \sqcup \exists_X(d_i))_{i \in \omega}$ that we know, by Propositions 5.2 and 5.4, to be a fair computation. It remains to prove that with $e_i = c \sqcup \exists_X(d_i)$ Conditions 1–4 are satisfied.

Conditions 1 and 2 follow immediately (remember that $\exists_X A$ is short for $\exists_X^{\perp} A$). When $i \in r(t)$, it follows that $\exists_X^{c_i} A_i : d_i \to \exists_X^{c_{i+1}} A_{i+1} : d_{i+1}$ and by the computation rules that $A_i : c \sqcup \exists_X(d_i) \to c_{i+1}$, and $d_{i+1} = d_i \sqcup \exists_X(c_{i+1})$. By the properties of the constraint system we have $d_{i+1} = d_{i+1} \sqcup \exists_X(d_{i+1}) = d_i \sqcup \exists_X(c_{i+1}) \sqcup \exists_X(d_{i+1}) = d_i \sqcup \exists_X(c_{i+1} \sqcup \exists_X(d_{i+1})) = d_i \sqcup \exists_X(e_{i+1})$. Condition 3 follows immediately.

If $i \in \omega \backslash r(t)$ the corresponding step in the computation $(\exists_X^{c_i} A_i : d_i)_{i \in \omega}$ is an input step. This implies that $c_i = c_{i+1}$. So $e_{i+1} = c_{i+1} \sqcup \exists_X(d_{i+1}) = c_i \sqcup \exists_X(d_{i+1}) = c_i \sqcup \exists_X(d_i) \sqcup \exists_X(d_{i+1}) = e_i \sqcup \exists_X(d_{i+1})$.

($\Leftarrow$) Assume that the right-hand side of the proposition holds. There is a computation $(A_i : e_i)_{i \in \omega}$ that connects the trace u to the agent A. For all $i \in \omega$, let $c_i = \bigsqcup \{e_{j+1} \mid j < i,\ j \in r(t)\}$ (this should agree with the idea that c_i, which is the local data of the agent, only changes when the agent performs computation steps).

To show that $(\exists_X^{c_i} A_i : d_i)_{i \in \omega}$ is a fair computation with trace t, note that for all $i \in \omega$, it follows that $\exists_X(d_i) = \exists_X(e_i)$ and $c_i \sqcup \exists_X(d_i) = e_i$. If $i \in r(t) = r(u)$, we know that $c_{i+1} = e_{i+1}$ and $A_i : e_i + A_{i+1} : e_{i+1}$. By the computation rules and the equalities above,

$$\exists_X^{c_i} A_i : d_i \to \exists_X^{c_{i+1}} A_{i+1} : d_{i+1}.$$

If $i \in \omega \backslash r(t)$ we have $A_{i+1} = A_i$ and $c_{i+1} = c_i$ so the ith step of $(\exists_X^{c_i} : A_i d_i)_{i \in \omega}$ is an input step.

To establish fairness of the computation $(\exists_X^{c_i} A_i : d_i)_{i \in \omega}$ it suffices to observe that its only immediate inner computation is fair. ▮

PROPOSITION 6.7. $t \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \ldots \,\square\, c_n \Rightarrow A_n)]\!]$ *iff one of the following holds.*

1. $c_j \sqsubseteq v(t)_k$ *for some* $j \leqslant n$ *and* $k \geqslant 0$, *and there is a* $u \in \mathcal{O}_\Pi[\![A_j]\!]$ *such that for all* $i \geqslant 0$, $v(u)_i = v(t)_{i+k+1}$, $v(t)_k = v(t)_{k+1}$, *and* $r(t) = \{i+k+1 \mid i \in r(u)\} \cup \{k\}$.

2. *There is no* $j \leqslant n$ *and* $k \geqslant 0$ *such that* $c_j \sqsubseteq v(t)_k$, *and* $r(t) = \varnothing$.

Proof. ($\Rightarrow$) Suppose $t \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \ldots \,\square\, c_n \Rightarrow A_n)]\!]$. Let $(B_i : d_i)_{i \in \omega}$ be the corresponding computation.

If $d_i \sqsupseteq c_l$, for some $i \in \omega$ and $l \leqslant n$, it follows by the fairness requirement that $B_{k+1} = A_j$ for some $k \geqslant 0$ and $j \leqslant n$. Since $(B_i : d_i)_{i \geqslant k+1}$ is a fair computation it follows that we have a corresponding trace $u \in \mathcal{O}_\Pi[\![A_j]\!]$. It is easy to see that the relationship between t and u are as stated in Condition 1.

If there is no $j \leqslant n$ and $i \in \omega$ such that $c_j \sqsubseteq d_i$, Condition 2 follows immediately.

($\Leftarrow$) Suppose Condition 1 holds. We will construct a fair computation $(B_i : d_i)_{i \in \omega}$ corresponding to the trace t. Let $(B_i : d_i)_{i \geqslant k+1}$ be the computation corresponding to the trace u. Let

$$B_0 = B_1 = \cdots = B_k = (c_1 \Rightarrow A_1 \,\square\, \ldots \,\square\, c_n \Rightarrow A_n),$$

and

$$d_0 = v(t_0),\ d_1 = v(t)_1,\ \ldots,\ d_k = v(t)_k.$$

It is straightforward to check that $(B_i : d_i)_{i \in \omega}$ is a computation, and that t is the corresponding trace. Fairness follows from the fact that a suffix is known to be fair.

In the case when Condition 2 holds, it is easy to check that we can construct a completely passive computation of the selection which has trace t. ∎

7. CLOSURE OPERATORS AND DETERMINISTIC PROGRAMS

Before we turn to the problem of giving a fully abstract semantics for ccp, we review some results from lattice theory concerning closure operators, and how closure operators can be used to give the semantics of certain concurrent constraint programs.

Jagadeesan *et al.* [15] showed how a concurrent process operating over a domain that allows "logic variables," i.e., place holders for values that are to be defined later, could be viewed as a closure operator. This idea was explored in a concurrent constraint programming setting by Saraswat *et al.* [32].

See reference [13] for further results on closure operators.

7.1. Closure Operators

Let us look at an agent as a function f that takes a store as input, and returns a new store. What properties are satisfied by the agent? First, the agent may never remove anything from the store, so the resulting store is always stronger than the original store. Thus, we have $f(x) \geqslant x$, for all x. Second, we assume that the process has finished all it wanted to do when it returned, thus applying it again will not change anything. Thus we have $f(f(x)) = f(x)$, for all x. Putting these two points together gives us the definition of closure operators.

DEFINITION 7.1. For a lattice $(D, \sqsubseteq)$, a *closure operator* over D is a monotone function f over D with the property that $f(x) \sqsupseteq x$ and $f(f(x)) = f(x)$, for any x in D. A *continuous closure operator* is a closure operator which is also continuous.

The set of fixpoints of a closure operator f over a lattice D is the set $f(D) = \{f(x) \mid x \in D\}$. Suppose that S is the set of fixpoints of a closure operator f, that is, $S = f(D)$, where D is the domain of f. For any subset T of S, $\bigsqcap T \in S$. This is easy to see if we observe that $f(\bigsqcap T) \sqsubseteq \bigsqcap \{f(x) \mid x \in T\}$, since f is monotone, $\bigsqcap T \sqsubseteq f(\bigsqcap T)$, since f is a closure operator, and $\bigsqcap \{f(x) \mid x \in T\} = \bigsqcap T$, since all members of T are fixpoints of f. It follows that the set of fixpoints of a closure operator is closed under greatest lower bounds of directed sets.

On the other hand, if $S \subseteq D$ is such that S is closed under arbitrary greatest lower bounds, we can define a function f_S according to the rule $f_S(x) = \bigsqcap(\{x\}^u \cap S)$, i.e., let $f_S(x)$ be the least element of S greater than x. It is easy to see that the function f_S is well-defined and a closure operator.

Thus there is a one-to-one correspondence between closure operators and sets closed under $\bigsqcap$. We take advantage of this property, and sometimes see closure operators as functions, and sometimes as sets. To say that x is a fixpoint of the closure operator f we can write $x = f(x)$ or $x \in f$.

Next we show that the closure operators over a lattice form a complete lattice. Consider the functions over a lattice to be ordered point-wise, i.e., $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x. Now we have $f \sqsubseteq g$ if and only if $f \sqsupseteq g$. If $\{f_i\}_{i \in I}$ is a family of closure operators, it is easy to see that $\bigcap_{i \in I} f_i$ is also a closure operator: this is obviously the least upper bound of the family of closure operators. The top element of the lattice of closure operators over a lattice D is the function that maps every element of the lattice to $\top$, and the bottom element is the identity function.

For an element $x \in D$ and a closure operator f over D, we define $(x \to f)$ as the closure operator given by

$$(x \to f)(y) = \begin{cases} f(y), & \text{if } \ y \sqsupseteq x \\ y, & \text{otherwise.} \end{cases}$$

Since a closure operator is characterized by its set of fixpoints, the following definition will also suffice.

$$(x \to f) = f \cup \{y \mid x \not\sqsupseteq y\}$$

Similarly, for elements $x \in D$ and $y \in D$ the closure operator $(x \to y)$ is defined as follows

$$(x \to y) = (x \to y\uparrow),$$

where $y\uparrow$ is the closure operator whose set of fixpoints is $\{y\}^u = \{z \in D \mid y \sqsubseteq z\}$. Note that when x is finite the closure operators $(x \to f)$ and $(x \to y)$ are continuous, for f continuous and arbitrary y.

Unless stated otherwise, we will assume the closure operators occurring in this paper to be continuous.

7.2. *Semantics of Deterministic ccp*

A *deterministic ccp program* is a ccp program in which all selections have only one alternative. Since non-determinism in ccp stems from selections in which more than one alternative can be selected, the restriction to selections with only one alternative effectively makes the programs deterministic. This is perhaps not completely obvious, since agents still execute concurrently, and results may be computed in different orders. However, as shown by Saraswat *et al.* [32], it turns out that the agents may be represented as functions with some special properties, and that the meaning of programs may be obtained as simple fixpoints, as in Kahn's semantics [18]. Since the fully abstract semantics in part relies on their semantics, we will review their results in this section.

First, consider a tell constraint c. Applying c to a store d gives us the store $c \sqcup d$. Thus the tell constraint c can be modeled with the closure operator $(\bot \to c)$.

To model a selection $(c \Rightarrow A)$, let us assume that the agent A can be modeled with the closure operator f. When examining the behavior of the selection, we see that it remains passive until the ask constraint c is entailed by the store, and then behaves like the agent A. Thus we can model the selection with the closure operator $(c \to f)$, which simply returns its input when c is not less that or equal to the input, and applies f to the input when the input is stronger than c.

To model a conjunction $A \wedge B$ (we only consider the finite conjunction, generalization is straight-forward), we assume that the semantics of A is the closure operator f, and that the semantics of the agent B is the closure operator g. Given a store c, the result of running the agent A is the new store $f(c)$. If we now run the agent B we obtain a new store $g(f(c))$. But now A can execute further and produce the store $f(g(f(c)))$. This can go on forever.

Suppose that A and B are allowed to interleave forever, thus producing the limit of the sequence of stores indicated above, what will the limit look like? It is easy to show by a mathematical argument (assuming that f and g are continuous) that the limit must be a fixpoint of both f and g. Also, the limit must be the smallest mutual fixpoint of f and g greater than c. Thus, if we want a function that models the behavior of $A \wedge B$, we should use the least closure operator stronger than f and g, which is $f \cap g$.

Consider an existential quantification $\exists_X A$, where the semantics of A is given by the closure operator f. If we run the agent $\exists_X A$ with a store c, the store accessible to A is given by $\exists_X(c)$. If the agent A produces a new store d, the part of the modification visible on the outside is $\exists_X(d)$. For example, if c entails the constraint $X = 10$, this aspect of c is not visible to A. If A chooses to add the tell constraint $X = 5$ to the store, this change is not visible to an outside observer.

Thus, we have a form of two-way hiding, and the semantics of $\exists_X A$ can be given by the closure operator g, given as

$$g(c) = c \sqcup \exists_X(f(\exists_X(c))).$$

To deal with the general case, we define a function E_X, which takes a function and returns the corresponding function where X is hidden. E_X can be defined as

$$\mathrm{E}_X(f) = \mathbf{id} \sqcup (\exists_X \circ f \circ \exists_X).$$

Now, if the semantics of A is f, the semantics of $\exists_X A$ is $\mathrm{E}_X(f)$.

We have seen that the basic constructs of determinate ccp can be modeled as continuous closure operators. It is time to write down the properties of the constructs in the form of a fixpoint semantics. We give a fixpoint semantics of deterministic ccp in which each agent is mapped to a continuous closure operator, and a program is mapped to a function from names to closure operators, i.e., an *environment*. Thus, the semantics for an agent A is given as a function $\mathscr{E}[\![A]\!]$: $(\mathscr{U} \to \mathscr{U})^{\mathscr{N}} \to (\mathscr{U} \to \mathscr{U})$, which maps environments to closure operators.

Now, to give the semantics of a call $p(X)$, we model the substitution using hiding and equality, so that the dummy variable α is used for argument-passing, just like in the operational semantics. Thus the semantic rule for procedure calls becomes

$$\mathscr{E}[\![p(X)]\!]\,\sigma = \mathrm{E}_\alpha((\alpha = X) \cap (\sigma p)).$$

In the same way, to give the semantics of the procedure definitions in a program Π, we define a function $\mathscr{P}[\![\Pi]\!]$ which takes a program and an environment and produces a "better" environment, as below. Assume that for each name p, the corresponding definition in Π is $p(Y) :: A$,

$$\mathscr{P}[\![\Pi]\!]\,\sigma p = \mathrm{E}_Y((Y = \alpha) \cap (\mathscr{E}[\![A]\!]\,\sigma)).$$

The semantics of a program Π is the least fixpoint of $\mathscr{P}[\![\Pi]\!]$. We can now put the fixpoint semantics together, as shown in Fig. 1.

Definition of $\mathcal{E}[\![A]\!] : (\mathcal{U} \to \mathcal{U})^{\mathcal{N}} \to (\mathcal{U} \to \mathcal{U})$

$$\begin{aligned}
\mathcal{E}[\![c]\!]\sigma &= (\bot \to c) \\
\mathcal{E}[\![\textstyle\bigwedge_{j\in I} A^j]\!]\sigma &= \bigcap_{j\in I} \mathcal{E}[\![A^j]\!]\sigma \\
\mathcal{E}[\![(c \Rightarrow A)]\!]\sigma &= (c \to \mathcal{E}[\![A]\!]\sigma) \\
\mathcal{E}[\![\exists_X A]\!]\sigma &= \mathrm{E}_X(\mathcal{E}[\![A]\!]\sigma) \\
\mathcal{E}[\![p(X)]\!]\sigma &= \mathrm{E}_\alpha((\alpha = X) \cap (\sigma p))
\end{aligned}$$

Definition of $\mathcal{P}[\![\Pi]\!] : (\mathcal{U} \to \mathcal{U})^{\mathcal{N}} \to (\mathcal{U} \to \mathcal{U})^{\mathcal{N}}$

$$\mathcal{P}[\![\Pi]\!]\sigma p = \mathrm{E}_Y((Y = \alpha) \cap (\mathcal{E}[\![A]\!]\sigma)),$$

where for each $p \in \mathcal{N}$ the definition in Π is assumed to be of the form $p(Y) :: A$, for some variable Y and some agent A

FIG. 1. The fixpoint semantics for determinate ccp.

8. A FULLY ABSTRACT SEMANTICS FOR NON-DETERMINISTIC CCP

We turn back to the problem of giving a fully abstract semantics for possibly non-deterministic ccp. The idea is that we look at two aspects of a trace; its functionality and its limit. The functionality can loosely be described as the closure operator computed by an agent in a computation which has this trace, and the limit is simply the limit of the sequence of stores of the trace.

DEFINITION 8.1. Let t be a trace. Let the *limit* of t be $\lim t = \bigcap_{i \in \omega} v(t)_i$. Let the *functionality* of t, denoted $\mathrm{fn}(t)$, be the closure operator

$$\mathrm{fn}(r) = \bigcap_{i \in r(t)} (v(t)_i \to v(t)_{i+1}).$$

Note that $\mathrm{fn}(t)$ is the least closure operator f such that

$$v(t)_{i+1} \sqsubseteq f(v(t)_i),$$

for all $i \in r(t)$.

We can characterise $\mathrm{fn}(t)$ in terms of its fixpoints.

PROPOSITION 8.2. *Let t be a trace. A constraint d is a fixpoint of* $\mathrm{fn}(t)$ *exactly when for all $i \in r(t)$, $d \sqsupseteq (v(t)_i)$ implies $d \sqsupseteq (v(t)_{i+1})$.*

The proposition follows from the definition of the functionality of a trace and from the properties of closure operators.

DEFINITION 8.3. We say that a trace t is a *subtrace* of a trace t', if the limit of t is equal to the limit of t' and the functionality of t is weaker than or equal to the functionality of t', i.e., $\mathrm{fn}(t) \supseteq \mathrm{fn}(t')$. A set $S \subseteq \mathrm{TRACE}$ is *subtrace-closed*, if $t \in S$ whenever t is a subtrace of t' and $t' \in S$.

DEFINITION 8.4. Given a trace t, the *inverse* of t is the trace $\bar{t} = (v(t), \omega \backslash r(t))$.

8.1. Definition of the Abstract Semantics.

DEFINITION 8.5. For an agent A, and a program Π, let

$$\mathscr{A}_\Pi[\![A]\!] = \{ t \mid t \text{ is a subtrace of } t', \text{ for some } t' \in \mathscr{O}_\Pi[\![A]\!] \}.$$

Not surprisingly, the abstract semantics contains sufficient information to allow the result semantics to be obtained from the abstract semantics.

PROPOSITION 8.6. *For an agent A, and constraint d, we have*

$$\mathscr{R}_\Pi[\![A]\!]\, d = \{ \lim(t) \mid t \in \mathscr{A}_\Pi[\![A]\!] \text{ and } \mathrm{fn}(t) = (d \to \lim(t)) \}.$$

The proposition follows from the operational semantics.

EXAMPLE 8.7. As an example of the abstract semantics, consider the following two agents. Let the agent A_1 be

$$X = [1, 2, 3, 4],$$

and the agent A_2 be

$$\exists_Y (X = [1, 2 \mid Y] \wedge Y = [3, 4]).$$

The two agents produce the same result, and there is no way a concurrently executing agent could see that the agent A_2 produces the list in two steps, so we would expect these two agents to have the same abstract semantics.

A typical trace of A_1 might be the trace t_1, where

$$v(t_1) = (\bot, X = [1, 2, 3, 4], \ldots)$$
$$r(t_1) = \{0 \quad \},$$

and a typical trace of A_2 might be the trace t_2, where

$$v(t_2) = (\bot, \exists_Y (X = [1, 2 \mid Y]), X = [1, 2, 3, 4], \ldots)$$
$$r(t_2) = \{0, \quad 1 \quad \}.$$

We also see that $\mathrm{fn}(t_1) = \mathrm{fn}(t_2) = (\bot \to X = [1, 2, 3, 4])$, and $\lim(t_1) = \lim(t_2) = (X = [1, 2, 3, 4])$. The two traces have the same functionality and limit and are thus subtraces of each other. Any trace of A_1 is a subtrace of some trace of A_2, and vice versa, so the two agents have the same abstract semantics.

8.2. Relationship with Deterministic Semantics

Recall that for deterministic ccp programs there is a simple fully abstract fixpoint semantics where the semantics of an agent is given as a closure operator (Section 7). Given a deterministic agent A and program Π, where the semantics of A is given by the closure operator f, what does the corresponding abstract semantics for A look like?

For finite constraints c and d, if $f(c) \sqsupseteq d$ it follows that A will, given a store where c holds, add constraints to the store so that d is entailed. If, on the other hand, $f(c) \not\sqsupseteq d$, we can conclude that if A starts executing with the store c, we will never arrive at a configuration where d is entailed (unless information is added from the outside). Thus the traces of A all have a functionality which is weaker or equal to that of f. The limit of any trace of A must be a constraint which is a fixpoint of f, otherwise the execution of A would have added more information to the store. Thus, the abstract semantics of A can be given as

$$\mathscr{A}_\Pi[\![A]\!] = \{t \mid \mathrm{fn}(t) \sqsubseteq f, \lim(t) \in f\}.$$

(The relationship is stated without proof since the further developments do not rely on it.)

8.3. Compositionality of the Abstract Semantics

The constructs that need to be considered are conjunction, the existential quantifier, and the selection operator.

8.3.1. *Conjunction.* The following lemma relates the result semantics of a conjunction of agents to the abstract semantics of the agents.

We take advantage of the fact that whenever $t \in \mathscr{O}_{\Pi}[\![A]\!]$, for some agent A, there is a trace t' satisfying $v(t')_0 = \perp$ with the same limit and functionality as t, defined by, e.g., $v(t')_0 = \perp$, $v(t')_{i+1} = v(t)_i$, and $i+1 \in r(t')$ iff $i \in r(t)$ for $i \in \omega$.

LEMMA 8.8. *Suppose $\{A^j\}_{j\in I}$ is a countable family of agents. For a constraint c, we have $c \in \mathscr{R}_{\Pi}[\![\bigwedge_{j\in I} A^j]\!]\perp$ if and only if there is a family of traces $(t_j)_{j\in I}$ such that $t_j \in \mathscr{A}_{\Pi}[\![A^j]\!]$ and $\lim(t_j) = c$ for $j \in I$, and $\bigcap_{j\in I} \mathrm{fn}(t_j) = c\uparrow$.*

Proof. ($\Rightarrow$) Suppose $c \in \mathscr{R}_{\Pi}[\![\bigwedge_{j\in I} A^j]\!]\perp$. By Proposition 8.6 there is an input-free trace $t \in \mathscr{O}[\![\bigwedge_{j\in I} A^j]\!]$ such that $v(t)_0 = \perp$, $\lim(t) = c$, and $\mathrm{fn}(t) = c\uparrow$. By Lemma 6.5 there is for each $j \in I$ a trace $t_j \in \mathscr{O}_{\Pi}[\![A^j]\!]$ such that $v(t_j) = v(t)$ and $\bigcup_{j\in I} r(t_j) = r(t)$. Using Proposition 8.2 it follows that a fixpoint of $\mathrm{fn}(t)$ is also a fixpoint of all $\mathrm{fn}(t_j)$, so $\bigcap_{j\in I} \mathrm{fn}(t_j) = c\uparrow$.

($\Leftarrow$) Each trace t_j connects A^j to a computation $(A^j_i : c^j_i)_{i\in\omega}$. We can assume that $c^j_0 = \perp$, for $j \in I$. We also assume that c is not finite. Let p be a function $p: \omega \to I$ such that for each $j \in I$ there are infinitely many $k \in \omega$ such that $p(k) = j$. We will form a computation $(B_i : d_i)_{i\in\omega}$ of the agent $\bigwedge_{j\in I} A^j$, where each B_i is of the form $\bigwedge_{j\in I} B^j_i$.

Let $B_0 = \bigwedge_{j\in I} A^j_0$. Let $d_0 = \bigsqcup_{j\in I} c^j_0$ ($= \perp$).

Suppose $B_k : d_k$ is defined for $k \leqslant n$. We define $B_{n+1} : d_{n+1}$ as follows. Let $k = p(n)$. Let m be the maximal integer such that $A^k_m = B^k_n$ and $c^k_m \sqsubseteq d_n$.

1. If there is a computation step $A^k_m : c^k_m \to A^k_{m+1} : c^k_{m+1}$, we make the constructed computation perform a corresponding computation step, by letting $B^j_{n+1} = B^j_n$, for $j \neq k$, $B^k_{n+1} = A^k_{m+1}$, and $d_{n+1} = d_n \sqcup c^k_{m+1}$.

2. If there is no computation step $A^k_m : c^k_m \to A^k_{m+1} : c^k_{m+1}$, let $B_{n+1} = B_n$ and $d_{n+1} = d_n$. Note that in this case d_n must be a fixpoint of $\mathrm{fn}(t_k)$ (since c^k_m is a fixpoint of $\mathrm{fn}(t_k)$, and by the way m was selected we know that $c^k_{m+1} \not\sqsubseteq d_n$).

Consider the limit $d = \bigsqcup_{n\in\omega} d_n$. Note that $d_n \sqsubseteq c$ for all n, so $d \sqsubseteq c$. Suppose $d \sqsubset c$. We can see that in the construction of $(B_n : d_n)_{n\in\omega}$, case 1 was only applied a finite number of times for each $j \in I$. This implies that for each $j \in I$, there is an infinite chain

$$d^j_0, d^j_1, d^j_2, \ldots$$

of fixpoints of $\mathrm{fn}(t_j)$. Since the limit of each of these chains is d, and by continuity, d must also be a fixpoint of each $\mathrm{fn}(t_j)$. But then d is a fixpoint of $\bigcap_{j\in I} \mathrm{fn}(t_j)$ and we arrive at a contradiction. ∎

From Lemma 8.8 it follows that the abstract semantics of a conjunction can be obtained from the agents. In the proof of the theorem below, we use the fact that

for an arbitrary trace, there is an agent whose operational semantics contains the trace. The construction of the agent is as follows.

DEFINITION 8.9. For a trace t, let $[t]$ be the agent

$$\bigwedge_{i \in r(t)} (v(t)_i \Rightarrow v(t)_{i+1}).$$

Clearly, $t \in \mathscr{A}_\Pi[\![[t]]\!]$. Moreover, we have the following:

LEMMA 8.10. *Let t be a trace. If $u \in \mathscr{O}_\Pi[\![[t]]\!]$ is a trace of $[t]$, then* $\mathrm{fn}(u) \sqsubseteq \mathrm{fn}(t)$.

The lemma follows from the computation rules.

LEMMA 8.11. *Let t be a trace such that $v(t)_0 = \bot$, let $\bar{t}$ be the inverse of t, and let $c = \lim(t)$. Then* $\mathrm{fn}(t) \cap \mathrm{fn}(\bar{t}) = c\uparrow$ *and* $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = \mathscr{U}$.

The lemma follows immediately from Proposition 8.2.

THEOREM 8.12. *Let $\{A^j \mid j \in I\}$ be a family of agents. For any trace t,*

$$t \in \mathscr{A}_\Pi\left[\!\!\left[\bigwedge_{j \in I} A^j\right]\!\!\right]$$

iff for each $j \in I$ there is a $t_j \in \mathscr{A}_\Pi[\![A^j]\!]$, such that $\lim(t) = \lim(t_j)$ *and* $\mathrm{fn}(t) \supseteq \bigcap_{j \in I} \mathrm{fn}(t_j)$.

Proof. ($\Rightarrow$) Let $t \in \mathscr{A}_\Pi[\![\bigwedge_{j \in I} A^j]\!]$. By definition there is a $t' \in \mathscr{O}_\Pi[\![\bigwedge_{j \in I} A^j]\!]$ such that t is a subtrace of t'. Let $c = \lim(t)$, and let B be the agent

$$B = \left(\bigwedge_{j \in I} A^j\right) \wedge [\bar{t}],$$

where $\bar{t}$ is the inverse of the trace t. Since $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = c\uparrow$, and $\mathrm{fn}(t') \subseteq \mathrm{fn}(t)$, and $c = \lim(t')$, we have by Lemma 8.8 that $c \in \mathscr{R}_\Pi[\![B]\!]\bot$. Again, by using the decomposition of B into $[\bar{t}]$ and the individual A^j in Lemma 8.8 it follows that there is a trace u of $[\bar{t}]$ and that for each $j \in I$ there is a $t_j \in \mathscr{O}_\Pi[\![A^j]\!]$, such that $\lim(t) = \lim(t_j)$ and $\bigcap_{j \in I} \mathrm{fn}(t_j) \cap \mathrm{fn}(u) = c\uparrow$. By $\mathrm{fn}(\bar{t}) \subseteq \mathrm{fn}(u)$ and $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = \mathscr{U}$ and $\mathrm{fn}(t) \cap \mathrm{fn}(\bar{t}) = c\uparrow$ we get $\mathrm{fn}(t) \supseteq \bigcap_{j \in I} \mathrm{fn}(t_j)$.

($\Leftarrow$) Suppose that for each $i \in I$ there is a $t_j \in \mathscr{A}_\Pi[\![A^j]\!]$ such that $\lim(t) = \lim(t_j)$ and $\mathrm{fn}(t) \supseteq \bigcap_{j \in I} \mathrm{fn}(t_j)$. By definition there is for each $j \in I$ a $t'_j \in \mathscr{O}_\Pi[\![A^j]\!]$ so that t_j is a subtrace of t'_j. Let B be the agent

$$B = \left(\bigwedge_{j \in I} A^j\right) \wedge [\bar{t}].$$

By $\mathrm{fn}(t) \supseteq \bigcap_{j \in I} \mathrm{fn}(t_j)$ and $\mathrm{fn}(t'_j) \subseteq \mathrm{fn}(t_j)$ and the fact that all involved traces have limit c, we infer that $\mathrm{fn}(t) \cap \bigcap_{j \in I} \mathrm{fn}(t'_j) = c\uparrow$. By Lemma 8.8 it follows that $c \in \mathscr{R}_\Pi[\![B]\!]\bot$. Again, by using the decomposition of B into $[\bar{t}]$ and the conjunction

$\bigwedge_{j\in I} A^j$ in Lemma 8.8 it follows that there is a trace t' of $\bigwedge_{j\in I} A^j$ and a trace u of $[\bar{t}]$ such that $\mathrm{fn}(t')\cap\mathrm{fn}(u)=c\uparrow$. Since $\mathrm{fn}(\bar{t})\subseteq\mathrm{fn}(u)$ and $\mathrm{fn}(t)\cup\mathrm{fn}(\bar{t})=\mathscr{U}$, we infer that $\mathrm{fn}(t')\subseteq\mathrm{fn}(t)$, i.e., that t is a subtrace of t' which implies that $t\in\mathscr{A}_{\Pi}[\![\bigwedge_{j\in I} A^j]\!]$. ▮

8.3.2. *The existential quantifier.* The treatment of the existential quantifier is certainly the most difficult part of this article. An early version of the article gave an incorrect characterization of the compositionality of the abstract semantics with respect to the existential quantifier. A similar error was made by Saraswat *et al.* [32]. When we look at the semantics of an existentially quantified agent $\exists_X A$, it should be clear that for any trace t of the agent $\exists_X A$ there is a corresponding trace u of the agent A. How are these two traces related? Obviously, since the variable X is hidden, the traces u and t need not agree on the behaviour with respect to X, but for other variables there should be a correspondence between the two traces. It follows that the limit and functionality of t and u should agree when we do not look at how the variable X is treated. Are these requirements sufficient? Well, almost. It turns out that it is necessary to add a third requirement to the trace u. For example, consider the agent

$$A=(X=10\Rightarrow Y=7\ \square\ \mathbf{true}\Rightarrow Z=5).$$

The agent is non-deterministic, since if $X=10$ it might either produce $Y=7$, or $Z=5$. However, the agent

$$\exists_X A$$

is *deterministic* and will always produce the result $Z=5$. In other words, when we give the semantics for $\exists_X A$, we should not consider the traces that have input steps which bind X.

Given that an agent A has a trace u, and that there is a corresponding trace t of $\exists_X A$, how are the functionalities of the traces related? Intuitively, the difference lies in that the functionality of t cannot depend on X, i.e., it cannot detect if X is bound or bind X to a value. These considerations suggest that we use the operation E_X, as defined in Section 7.

PROPOSITION 8.13. *For a closure operator f, the closure operator $E_X(f)$ has the set of fixpoints given by*

$$E_X(f)=\{c\mid \textit{There is a constraint } d\in f \textit{ such that } \exists_X(c)=\exists_X(d)\}.$$

Proof. Note that for any constraint c, we have $E_X(f)\,c\sqsubseteq f(c)$. From this follows that any fixpoint of f must also be a fixpoint of $E_X(f)$. Let $g=E_X(f)$.

$(\supseteq)$ Suppose we have a constraint $d\in f$. It follows that $d=g(d)$. Let c be a constraint such that $\exists_X(c)=\exists_X(d)$. Applying g gives $g(c)=\exists_X(f(\exists_X(c)))\sqcup c=\exists_X(f(\exists_X(d)))\sqcup c\sqsubseteq\exists_X(f(d))\sqcup c=\exists_X(d)\sqcup c=\exists_X(c)\sqcup c=c$.

($\subseteq$) Now, suppose that c is a fixpoint of g. Let $d = f(\exists_X(c))$. The constraint d is of course a fixpoint of f and since $c = g(c)$, we must have $c \sqsupseteq \exists_X(f(\exists_X(c))) = \exists_X(d)$. So $\exists_X(c) \sqsupseteq \exists_X(d)$, and since $f(\exists_X(c)) \sqsupseteq \exists_X(c)$, which implies $\exists_X(c) \sqsubseteq \exists_X(d)$, we have $\exists_X(c) = \exists_X(d)$. ∎

In the proof of the compositionality theorem, the following proposition is useful. Note that for a trace t, the traces of $\mathscr{A}_\Pi[\![\exists_X[t]]\!]$ are the traces which have a functionality weaker than the one of t and a limit which is a fixpoint of the functionality of t.

PROPOSITION 8.14. *Given an agent A, let $u \in \mathscr{A}_\Pi[\![A]\!]$ such that* $(\mathrm{fn}\, u)(\exists_X(\lim u)) = \lim u$. *Let t be a trace such that* $\mathrm{fn}\, t \sqsubseteq \mathrm{E}_X(\mathrm{fn}\, u)$ *and* $\exists_X(\lim t) = \exists_X(\lim u)$. *It follows that* $\lim u \in \mathscr{R}_\Pi[\![A \wedge \exists_X[\bar{t}]]\!]$.

Proof. Note that $u \in \mathscr{A}_\Pi[\![A]\!]$ and there is a trace $t_0 \in \mathscr{A}_\Pi[\![\exists_X[\bar{t}]]\!]$ such that $\mathrm{fn}\, t_0 = \mathrm{E}_X(\mathrm{fn}\, \bar{t})$ and $\lim u = \lim t_0$. By Lemma 8.8 it is sufficient to show that $(\mathrm{fn}\, u) \cap (\mathrm{E}_X(\mathrm{fn}\, \bar{t}) = (\bot \to \lim u)$. We compute the least fixed point of $(\mathrm{fn}\, u) \cap (\mathrm{E}_X(\mathrm{fn}\, \bar{t}))$ by forming the chains $d_0, d_1, d_2, \ldots$ and $e_0, e_1, e_2, \ldots$ as

1. $d_0 = e_0 = \bot$.
2. For i even, let
   (a) $d_{i+1} = \mathrm{E}_X(\mathrm{fn}\, u)\, d_i$, and
   (b) $e_{i+1} = (\mathrm{fn}\, u)\, e_i$.
3. For i odd, let
   (a) $d_{i+1} = (\mathrm{fn}\, \bar{t})\, d_i$ and
   (b) $e_{i+1} = \mathrm{E}_X(\mathrm{fn}\, \bar{t})\, e_i$.

It is straightforward to show that $\exists_X d_i = \exists_X e_i$, for all $i \in \omega$. Let $d = \bigsqcup_{i \in \omega} d_i$ and $e = \bigsqcup_{i \in \omega} e_i$. We want to show that $d = \lim t$. Suppose $d \sqsubset \lim t$. By continuity, $d \in \mathrm{fn}\, \bar{t}$ and $d \in \mathrm{E}_X(\mathrm{fn}\, u)$. Thus $d \notin \mathrm{fn}\, t$ which implies that $d \notin \mathrm{E}_X(\mathrm{fn}\, u)$. We have arrived at a contradiction.

It follows that $d = \lim t$. By assumption we have $\exists_X(\lim t) = \exists_X(\lim u)$ and $(\mathrm{fn}\, u)(\exists_X(\lim u)) = \lim u$. It follows immediately that $(\mathrm{fn}\, u)(\exists_X d) = \lim u$. We can conclude that $e = (\mathrm{fn}\, u)(\exists_X d) = \lim u$. ∎

THEOREM 8.15. *For an agent A and a variable X there is a trace $t \in \mathscr{A}_\Pi[\![\exists_X A]\!]$ iff there is a $u \in \mathscr{A}_\Pi[\![A]\!]$ such that* $\lim(u) = (\mathrm{fn}(u) \circ \exists_X)\ \lim(u)$, $\exists_X(\lim(t)) = \exists_X(\lim(u))$ *and* $\mathrm{fn}(t) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$.

Proof. ($\Rightarrow$) Suppose $t \in \mathscr{O}_\Pi[\![\exists_X A]\!]$. By Proposition 6.6, there must be a trace $u \in \mathscr{O}_\Pi[\![A]\!]$, such that with $v(t) = (d_i)_{i \in \omega}$ and $v(u) = (e_i)_{i \in \omega}$ it holds that $r(t) = r(u)$; $e_0 = \exists_X(d_0)$; $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, for $i \in r(t)$; and $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$, for $i \in \omega \backslash r(t)$. It is straightforward to prove by induction that for all $i \in \omega$, $\exists_X(d_i) = \exists_X(e_i)$, and thus, $\exists_X(\lim(t)) = \exists_X(\lim(u))$.

Next we show that $\mathrm{fn}(t) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$. Let i be fixed such that $i \in r(t)$. It is sufficient to show that $(d_i \to d_{i+1}) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$. Note that for all $i \in \omega$, $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i))$ (this is easily proved by induction). If c is a constraint such that

$c \sqsupseteq d_i$, we have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i)) \sqsubseteq \mathrm{fn}(u)(\exists_X(c))$, and thus $\mathrm{fn}(u)(\exists_X(c)) \sqsupseteq e_{i+1}$, from which follows that $\mathrm{E}_X(\mathrm{fn}(u))\, c \sqsupseteq d_{i+1}$, since by the reduction rules $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$.

To show that $\lim(u) = (\mathrm{fn}(u) \circ \exists_X) \lim(u)$, we first note that $\mathrm{fn}(u)\, e_i \sqsubseteq \lim(u)$, for all $i \in \omega$, from which follows that $(\mathrm{fn}(u) \circ \exists_X)\, e_i \sqsubseteq \lim(u)$, for all i, and thus $(\mathrm{fn}(u) \circ \exists_X) \lim(u) \sqsubseteq \lim(u)$. By the argument in the previous paragraph we have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i))$, for $i \in \omega$, and since $\exists_X(d_i) = \exists_X(e_i)$ we also have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(e_i))$, for all i. By continuity we have $\lim(u) \sqsubseteq \mathrm{fn}(u)(\exists_X(\lim(u)))$.

($\Leftarrow$) Suppose that $u \in \mathscr{O}_\Pi[\![A]\!]$ such that $\lim(u) = (\mathrm{fn}(u) \circ \exists_X) \lim(u)$. Suppose also that the trace t is such that $\lim t = \lim u$ and $\mathrm{fn}\, t \sqsubseteq \mathrm{E}_X(\mathrm{fn}\, u)$. We want to show that $t \in \mathscr{A}_\Pi[\![\exists_X A]\!]$.

By the Proposition 8.14 there is a fair, input-free computation $(A_i \wedge \exists_X^{c_i} B_i : e_i)_{i \in \omega}$ where the configuration $A_0 \wedge \exists_X^{c_0} B_0 : e_0$ is equal to $A \wedge \exists_X[\bar{t}] : \bot$ and $\bigsqcup_{i \in \omega} e_i = \lim u$. Let u' be the trace corresponding to the computation $(A_i : e_i)_{i \in \omega}$. We have, by the computation rules,

1. $A_i : e_i \to A_{i+1} : e_{i+1}$ and $B_i : c_i = B_{i+1} : c_{i+1}$, if $i \in r(u')$, and
2. $B_i : c_i \sqcup (\exists_X e_i) \to B_{i+1} : c_{i+1}$, $A_{i+1} = A_i$ and $e_{i+1} = e_i \sqcup (\exists_X c_{i+1})$ if $i \notin r(u')$.

Let the chain $d_0, d_1, \ldots$ be as follows.

1. $d_0 = \bot$.
2. $d_{i+1} = c_{i+1} \sqcup \exists_X e_{i+1}$, if $i \in r(u)$.
3. $d_{i+1} = c_{i+1}$, if $i \notin r(u)$.

It is straightforward to establish that for $i \in r(u)$, $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, and for $i \notin r(u)$, $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$. We can now form a trace t' with $r(t') = r(u)$ and $v(t') = (d_i)_{i \in \omega}$ such that, by Proposition 6.6, $t' \in \mathscr{O}_\Pi[\![\exists_X A]\!]$.

We also would like to show that t is a subtrace of t'. Consider the computation $(B_i : c_i \sqcup \exists_X e_i)_{i \in \omega}$ of $[\bar{t}]$. Clearly this is the same as $(B_i : d_i \sqcup \exists_X e_i)_{i \in \omega}$. Note that the trace of this computation is $\bar{t}'$. It follows that $\mathrm{fn}\, t' \sqsupseteq \mathrm{fn}\, t$ and that $\lim t' = \lim t$. ∎

8.3.3. *The selection operator.*

THEOREM 8.16. *For $n \geqslant 0$, agents $A_1, \ldots, A_n$ and constraints $c_1, \ldots, c_n$, we have a trace $t \in \mathscr{A}_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$ if and only if either*

1. *there is a $k \leqslant n$ and $u \in A_\Pi[\![A_k]\!]$ such that $\lim(t) = \lim(u) \sqsupseteq c_k$ and $\mathrm{fn}(t) \sqsubseteq (c_k \to \mathrm{fn}(u))$, or*
2. *$\mathrm{fn}(t) = \mathbf{id}$ and $c_k \not\sqsubseteq \lim(t)$, for $k \leqslant n$.*

Proof. ($\Rightarrow$) Suppose $t \in A_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$. There is a trace $t' \in \mathscr{O}_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$ such that t is a subtrace of t'.

Suppose that $\lim(t) \sqsupseteq c_l$, for some $l \leqslant n$. By Proposition 6.7 there is, for some $k \leqslant n$, a trace $u \in \mathscr{O}_\Pi[\![A_k]\!]$ such that $\lim(u') = \lim(t')$ and $\mathrm{fn}(t') = (c_k \to \mathrm{fn}(u))$. The trace u is of course also a trace of $A_\Pi[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$. Since $\mathrm{fn}(t) \sqsubseteq \mathrm{fn}(t)$, we have $\mathrm{fn}(t) \sqsubseteq (c_k \to \mathrm{fn}(u))$.

Suppose that there are no $l \leqslant n$ such that $\lim(t) \sqsupseteq c_l$. By Proposition 6.7 we have $r(t) = \varnothing$, and thus $\mathrm{fn}(t) = \mathbf{id}$.

($\Leftarrow$) Suppose $u \in \mathscr{A}_{\Pi}[\![A_k]\!]$, and that t is a trace such that $\lim(t) = \lim(u) \sqsupseteq c_k$, and $\mathrm{fn}(t) \sqsubseteq (c_k \to \mathrm{fn}(u))$. We have immediately that u is a subtrace of a trace $u' \in \mathscr{O}_{\Pi} A_k$. By Proposition 6.7 there is a trace $t' \in \mathscr{O}[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$ such that $\lim(t') = \lim(u')$ and $\mathrm{fn}(t') = (c_k \to \mathrm{fn}(u'))$. Thus, $\mathrm{fn}(t') \sqsupseteq (c_k \to \mathrm{fn}(u)) \sqsupseteq \mathrm{fn}(t)$, and we conclude t is a subtrace of t'.

Let t be a trace such that there are no $l \leqslant n$ such that $\lim(t) \sqsupseteq c_l$. By Proposition 6.7 it follows that $t \in \mathscr{O}_{\Pi}[\![(c_1 \Rightarrow A_1 \,\square\, \cdots \,\square\, c_n \Rightarrow A_n)]\!]$, and thus t also belongs to the abstract semantics of the selection. ▮

8.4. The Abstract Semantics in Equational Form

As we have shown that the abstract semantics is compositional, we summarise the semantic equations in Fig. 2.

8.5. The Trace Semantics is Fully Abstract

A semantics is *fully abstract* if the semantics does not give more information than what is necessary to distinguish between agents that behave differently when put into a context. The following theorem states that the trace semantics is indeed fully abstract.

THEOREM 8.17 (Full Abstraction). *Suppose we have agents A and A'. If $\mathscr{A}_{\Pi}[\![A]\!] \neq \mathscr{A}_{\Pi}[\![A']\!]$ then there is an agent B such that $\mathscr{R}_{\Pi}[\![A \wedge B]\!] \neq \mathscr{R}_{\Pi}[\![A' \wedge B]\!]$.*

Proof. Suppose $t \in \mathscr{A}_{\Pi}[\![A]\!] \backslash \mathscr{A}_{\Pi}[\![A']\!]$. Consider the agent $A \wedge [\bar{t}]$. By Lemma 8.8 we have $\lim(t) \in \mathscr{R}_{\Pi}[\![A \wedge [\bar{t}]]\!]\bot$. Suppose $\lim(t) \in \mathscr{R}_{\Pi}[\![A' \wedge [\bar{t}]]\!]\bot$. By Lemma 8.8 there are traces $t_1 \in \mathscr{O}_{\Pi}[\![A']\!]$ and $t_2 \in \mathscr{O}_{\Pi}[\![[\bar{t}]]\!]$ such that $\lim(t_1) = \lim(t_2) = \lim(t)$ and $\mathrm{fn}(t_1) \cap \mathrm{fn}(t_2) = c\uparrow$. Since clearly $\mathrm{fn}(t_2) \supseteq \mathrm{fn}(\bar{t})$, this implies that $\mathrm{fn}(t_1) \subseteq \mathrm{fn}(t)$, so t must be a subtrace of t_1. This contradicts the assumption that $t \notin \mathscr{O}_{\Pi}[\![A']\!]$. ▮

$$
\begin{aligned}
\mathscr{A}_{\Pi}[\![c]\!] &= \{t \mid \mathrm{fn}(t) \supseteq (\bot \to c) \text{ and } \lim(t) \sqsupseteq c\} \\
\mathscr{A}_{\Pi}[\![\textstyle\bigwedge_{j \in I} A^j]\!] &= \{t \mid t_j \in \mathscr{A}_{\Pi}[\![A^j]\!] \text{ and } \lim(t_j) = \lim(t), \text{ for } j \in I, \\
&\qquad \mathrm{fn}(t) \supseteq \textstyle\bigcap_{j \in I} \mathrm{fn}(t_j)\} \\
\mathscr{A}_{\Pi}[\![\exists_X A]\!] &= \{t \mid t' \in \mathscr{A}_{\Pi}[\![A]\!], \\
&\qquad \exists_X(\lim(t)) = \exists_X(\lim(t')), \\
&\qquad \lim(t') = (\mathrm{fn}(t') \circ \exists_X)\lim(t'), \text{ and} \\
&\qquad \mathrm{fn}(t) \supseteq \mathrm{E}_X(\mathrm{fn}(t'))\} \\
\mathscr{A}_{\Pi}[\![\square_{k \leq n} c_k \Rightarrow A_k]\!] &= \{t \mid \mathrm{fn}(t) = \mathbf{id} \text{ and } \lim(t) \not\sqsupseteq c_k, \text{ for } k \leq n\} \bigcup \\
&\qquad \{t \mid k \leq n, t' \in \mathscr{A}_{\Pi}[\![A_k]\!], \\
&\qquad \lim(t) = \lim(t') \sqsupseteq c_k, \text{ and} \\
&\qquad \mathrm{fn}(t) = (c_k \to \mathrm{fn}(t'))\} \\
\mathscr{A}_{\Pi}[\![p(X)]\!] &= \mathscr{A}_{\Pi}[\![A[X/Y]]\!], \\
&\qquad \text{where the definition of } p \text{ is } p(Y) :: A
\end{aligned}
$$

FIG. 2. The abstract semantics in equational form.

8.6. Examples of Abstract Semantics

We give abstract semantics of some concurrent constraint programs. The first four of the programs were given in Section 3.1. The programs assume the term model, defined in Example 2.5.

EXAMPLE 8.18 (Non-determinism). We consider the "erratic" program given in Example 3.1. The abstract semantics is as follows.

$t \in \mathscr{A}_{\Pi}[\![\text{erratic}(X)]\!]$ if and only if either

1. t is such that $\lim(t) \sqsupseteq (X=0)$ and $\text{fn}(t) \sqsubseteq (\bot \to X=0)$, or
2. t is such that $\lim(t) \sqsupseteq (X=1)$ and $\text{fn}(t) \sqsubseteq (\bot \to X=1)$.

Note how the traces fall into two groups due to the non-determinism of the agent.

EXAMPLE 8.19 (McCarthy's ambiguity operator). The second program we consider is an implementation of McCarthy's ambiguity operator, given in Example 3.2. The abstract semantics is as follows.

$t \in \mathscr{A}_{\Pi}[\![\text{amb}(X, Y, Z)]\!]$ if and only if either

1. t is such that $\text{fn}(t) = \mathbf{id}$, and $\lim(t) \not\sqsupseteq \text{number}(X)$, and $\lim(t) \not\sqsupseteq \text{number}(Y)$, or
2. t is such that $\lim(t) \sqsupseteq \text{number}(X)$ and $\text{fn}(t) \sqsubseteq (\bot \to Z=X)$, or
3. t is such that $\lim(t) \sqsupseteq \text{number}(Y)$ and $\text{fn}(t) \sqsubseteq (\bot \to Z=Y)$.

EXAMPLE 8.20 (The "ones" program). Consider as a simple example of a program that produces an infinite result the "ones" program as given in Example 3.3.

For a trace t we have $t \in \mathscr{A}_{\Pi}[\![\text{ones}(X)]\!]$ if and only if t is such that $\lim(t) \sqsupseteq (X=1^{\omega})$ and $\text{fn}(t) \sqsubseteq (\bot \to 1^{\omega})$, where 1^{ω} is the infinite list of ones.

EXAMPLE 8.21 (lazy_ones). Consider the "lazy_ones" program in Example 3.4. We write $|X| \geqslant k$ for the constraint which says that X is a list with at least k elements, and $X[k]=1$ for the constraint that X is a list with at least k elements, and the kth is equal to 1.

The abstract semantics of a call lazy_ones(X) is the set of traces t for which

1. $\text{fn}(t) \sqsubseteq \bigcap_{k \geqslant 0} (|X| \geqslant k \to X[k]=1)$ and
2. (a) either $\lim(t) \sqsupseteq (X=1^{\omega})$, or (b) there is an $n \geqslant 0$ such that $\lim(t) \sqsupseteq \bigsqcup_{k<n} X[k]=1$ but $\lim(t) \not\sqsupseteq (|X| \geqslant n)$.

EXAMPLE 8.22 (The Keller–Brock–Ackerman anomaly). One proposed solution to the problem of giving a denotational semantics of a concurrent programming language was to map each process (or agent) to an input-output relation. However, as shown by Keller [19] and Brock and Ackerman [5], a semantics of this type

will not be compositional. The proof gives two programs with the same input-output relation but with different behaviour when put in a context. In this section, we give a similar example (the example is of a type first described by Russell [31]). Consider the two programs

$$\begin{aligned}&\mathrm{m1}(X, Y, R, Z) ::\\ &\quad \exists_{Z1}(\mathrm{number}(X) \Rightarrow Z1 = X \;\square\; \mathrm{number}(Y) \Rightarrow Z1 = Y) \wedge\\ &\quad\quad (\mathrm{number}(Z1) \Rightarrow R = 1) \wedge\\ &\quad\quad (\mathrm{number}(X) \wedge \mathrm{number}(Y) \Rightarrow Z = Z1)\end{aligned}$$

and

$$\begin{aligned}&\mathrm{m2}(X, Y, R, Z) ::\\ &\quad\quad (\mathrm{number}(X) \Rightarrow R = 1) \wedge\\ &\quad\quad (\mathrm{number}(Y) \Rightarrow R = 1) \wedge\\ &\quad\quad (\mathrm{number}(X) \wedge \mathrm{number}(Y) \Rightarrow Z = X \;\square\\ &\quad\quad\; \mathrm{number}(X) \wedge \mathrm{number}(Y) \Rightarrow Z = Y).\end{aligned}$$

The agents $A_1 = \mathrm{m1}(X, Y, R, Z)$ and $A_2 = \mathrm{m2}(X, Y, R, Z)$ have the same result semantics. Both agents wait until either X or Y is constrained to be a number, and then bind R to 1. When both X and Y are bound to numbers, Z is bound to either X or Y. For example, given an initial state $c = (X = 42)$, the result semantics of A_1 and A_2 is

$$\mathscr{R}_{\Pi}[\![A_1]\!]\, c = \mathscr{R}_{\Pi}[\![A_2]\!]\, c = \{c \wedge (R = 1)\}.$$

If we assume an initial state $d = (X = 42 \wedge Y = 43)$ the result semantics is

$$\mathscr{R}_{\Pi}[\![A_1]\!]\, d = \mathscr{R}_{\Pi}[\![A_2]\!]\, d = \{(d \wedge R = 1 \wedge Z = 42), (d \wedge R = 1 \wedge Z = 43)\}.$$

The two programs differ in that m1 records which variable was bound first, so that if Y was not bound until $R = 1$, we know that Z will be bound to X.

The abstract semantics of procedure m1 contains traces with functionality given by the closure operator

$$(\mathrm{number}(X) \to R = 1) \cap (\mathrm{number}(X) \sqcup \mathrm{number}(Y) \sqcup (R = 1) \to Z = X)$$

but no traces with the functionality

$$(\mathrm{number}(X) \to R = 1) \cap (\mathrm{number}(X) \sqcup \mathrm{number}(Y) \sqcup (R = 1) \to Z = Y)$$

but procedure m2 contains traces of both types. To detect the difference in semantics between m1 and m2, one can run them in conjunction with an agent

$$\text{test}(X, Y, R) :: X = 42 \wedge (R = 1 \Rightarrow Y = 43)$$

that binds X, waits until R is equal to 1, and then binds Y. The agent

$$\text{test}(X, Y, R) \wedge \text{m1}(X, Y, R, Z)$$

will always bind Z to 42, but the agent

$$\text{test}(X, Y, R) \wedge \text{m2}(X, Y, R, Z)$$

may bind Z to 43.

9. A PROOF OF FULL ABSTRACTION USING FINITE PROGRAMS

Our proof of full abstraction in Theorem 8.17 relied on the use of infinite conjunctions to express an agent that could produce an "infinite" trace. Is it possible to give a proof of full abstraction that does not use infinite conjunctions? It turns out that if we make some very reasonable assumptions about the constraint system, and extend the result semantics to cope with infinite input, it is possible to give a proof of full abstraction that does not use infinite conjunctions.

The proof in this section resembles a proof of full abstraction given by Russell [31] for data-flow networks. The idea is that we assume that a representation of a trace is provided as input. It is then possible to write a procedure that "interprets" the trace and thus exhibits a behaviour similar to the agent $[\bar{t}]$ in the previous proof. We must make some assumptions about the constraint system. First, we assume that in the domain of values there is a representation of each finite constraint. We also assume that it is possible to write procedures that can take a representation of a finite constraint and can interpret its behaviour. Third, we assume that the term model is a part of the constraint system and that there are some appropriate function symbols.

9.1. The Generalised Result Semantics

We previously defined the result semantics of an agent, $\mathscr{R}_{\Pi}[\![A]\!]\, c$, only for finite inputs c. This restriction was introduced to make it possible to give the input in the first configuration of a computation, as any intermediate state of a computation must be finite. However, if we allow the input to be given during the course of a computation, we can consider a generalised version of the result semantics that also allows infinite inputs.

For an agent A, a program Π, and a constraint c, the *generalised result semantics* is

$$\begin{aligned}\mathscr{R}_{\Pi}[\![A]\!]\, c = \{\lim(t) \mid\ & t \in \mathscr{C}_{\Pi}[\![A]\!],\\ & v(t)_0 \sqsubseteq c,\\ & c_{i+1} \sqsubseteq c_i \sqcup c, \text{ for } i \in \omega \backslash r(t), \text{ and}\\ & \lim(t) \sqsupseteq c\}.\end{aligned}$$

Clearly, for finite constraints the generalised result semantics conforms with the first result semantics. For infinite constraints the generalised result semantics gives the result produced by an agent when it receives infinite input.

PROPOSITION 9.1. *For an agent A, and constraints c and d, we have $c \in \mathscr{R}_{\Pi}[\![A]\!]\, d$ iff there is a trace $t \in \mathscr{A}_{\Pi}[\![A]\!]$ such that $\lim(t) = c$, and $\mathrm{fn}(t) \cap (\bot \to d) = (\bot \to c)$.*

9.2. Can a ccp Language Interpret its Constraint System?

The answer, for any reasonable constraint system, is yes. But first we must define what it means for a constraint system to be self-interpretable.

First, there must be a way to represent the finite constraints as values in the constraint system. For example, in the term model we can of course represent the finite constraints as terms.

Second, for each finite constraint we need a way to bind a variable to that finite constraint, i.e., a finite constraint that does precisely that.

Of course, we also need to be able to interpret the representations of constraints as real constraints, i.e., as tell and ask constraints. So we also require that it is possible to write procedures that interpret representations of constraints and emulate the behaviour of the corresponding ask and tell constraints (these are requirements three and four). In the term model, implementing these procedures is a straightforward programming task.

DEFINITION 9.2. A constraint system is *self-interpretable* if the following holds.

1. There is an injective map l from finite constraints to the domain of values.

2. For each finite constraint c and variable X there is a constraint $X = l(c)$ which binds X to $l(c)$.

3. For variables $X_1, ..., X_n$ it is possible to define a procedure entail such that a call entail $(R, F, X_1, ..., X_n)$ will bind the variable F to 1 provided that $R = l(c)$, where c is a constraint which is entailed by the store and only depends on variables $X_1, ..., X_n$.

4. For variables $X_1, ..., X_n$ it is possible to define a procedure toStore such that a call toStore$(R, X_1, ..., X_n)$ will add the constraint c to the store, provided that $R = l(c)$ and c is a constraint that depends only on the variables $X_1, ..., X_n$.

One would expect a constraint system to be implementable on a computer, and to be sufficiently powerful to implement the set of computable functions. Given this, it is not a big step to assume a constraint system to be self-interpretable. To represent the finite constraints it is of course very natural to use the elements of the term model.

9.3. Giving the Representation of a Trace

We need a way to construct a constraint that gives a representation of a trace. Suppose that t is a trace which only depends on variables $X_1, ..., X_n$. We must construct a constraint c (we will refer to it as $[t]$) which binds a variable L to a list representation of the trace t. Let $(d_i)_{i\in\omega} = v(t)$. Let E_0 be in (Z_0). For $i-1 \in r(t)$, let E_i be the expression $\text{out}(Z_i)$, and for $i-1 \in \omega\backslash r(t)$, let E_i be $\text{in}(Z_i)$. For $i \in \omega$, let

$$c_i \quad \text{be} \quad \exists_{L'} \exists_{Z_0} \cdots \exists_{Z_i} (L = [E_0, ..., E_i \mid L'] \wedge Z_0 = l(d_0) \wedge \cdots \wedge Z_i = l(d_i)).$$

Clearly, all c_is are finite constraints, and with $c = \bigsqcup_{i\in\omega} c_i$, c is the constraint which binds L to a representation of the trace t.

9.4. Interpreting Traces

Next we construct a procedure $\text{interpret}(L, X_1, ..., X_n)$, that given a list representation of a trace t that only depends on the variables $X_1, ..., X_n$, behaves like the agent $[t]$ in the previous proof of full abstraction. We want the call $\text{interpret}(L, X_1, ..., X_n)$ to be such that for traces

$$u \in \mathscr{A}_{\Pi}[\![\text{interpret}(L, X_1, ..., X_n)]\!]$$

we have $\text{fn}(u) \sqsubseteq \text{fn}(t)$ whenever $\exists_L(\text{lim}(u)) = \text{lim}(t)$ and L is bound to the list representation in the constraint $\text{lim}(u)$.

We first give interpret in the form of a clp program, since this version may be easier to read than the ccp version.

$$\begin{aligned}
&\text{interpret}([\text{out}(R) \mid L], X_1, ..., X_n) : - \\
&\qquad \text{toStore}(R, X_1, ..., X_n), \\
&\qquad \text{interpret}(L, X_1, ..., X_n). \\
&\text{interpret}([\text{in}(R) \mid L], X_1, ..., X_n) : - \\
&\qquad \text{entail}(R, F, X_1, ..., X_n), \\
&\qquad \text{interpret_aux}(F, L, X_1, ..., X_n). \\
&\text{interpret_aux}(1, L, X_1, ..., X_n) : - \text{interpret}(L, X_1, ..., X_n).
\end{aligned}$$

The same program in ccp,

$$
\begin{aligned}
&\text{interpret}(L, X_1, ..., X_n) ::\\
&\quad((\exists_R \exists_{L'} L = [\text{out}(R) \mid L'])\\
&\qquad \Rightarrow \exists_R \exists_{L'} (L = [\text{out}(R) \mid L']\\
&\qquad\qquad \wedge \text{toStore}(R, X_1, ..., X_n)\\
&\qquad\qquad \wedge \text{interpret}(L', X_1, ..., X_n))\\
&\quad \square\, \exists_R \exists_{L'} (L = [\text{in}(R) \mid L'])\\
&\qquad \Rightarrow \exists_R \exists_{L'} \exists_F (L = [\text{in}(R) \mid L'])\\
&\qquad\qquad \wedge \text{entail}(R, F, X_1, ..., X_n)\\
&\qquad\qquad \wedge (F = 1 \Rightarrow \text{interpret}(L', X_1, ..., X_n)))).
\end{aligned}
$$

Does interpret behave as intended? Suppose that L does eventually get bound to the list representation of the trace t. The functionality of any trace of the call interpret$(L, X_1, ..., X_n)$ is at most f_0, where f_i, $i \in \omega$, is given as follows (recall that $(d_i)_{i\in\omega} = v(t)$).

$$
f_i = \begin{cases} f_{i+1}, & \text{if} \quad i \notin r(t) \\ (d_i \to d_{i+1}) \cap f_{i+1}, & \text{if} \quad i \in r(t) \end{cases}
$$

It is easy to establish that $f_0 = \text{fn}(t)$.

9.5. *The Alternative Proof of Full Abstraction*

We will assume that the constraint system is self-interpretable, and contains the term model, where the set of function symbols includes the list constructor $[\cdot|\cdot]$, in$(\cdot)$ and out$(\cdot)$. Recall that A and A' are assumed to be agents such that $t \in \mathscr{A}_\Pi[\![A]\!]$ but $t \notin \mathscr{A}_\Pi[\![A']\!]$. We want to show that there is an agent B and some constraint c such that $\mathscr{R}_\Pi[\![A \wedge B]\!]\, c \neq \mathscr{R}_\Pi[\![A' \wedge B]\!]\, c$. We will assume that the agents A and A' do not contain any infinite conjunctions, and that thus the set of variables that A and A' depend on are among $X_1, ..., X_n$, and that L is not among these variables.

Let c be the constraint $[\bar{t}]$. Let B be the agent interpret$(L, X_1, ..., X_n)$ and Π' the program Π extended with definitions of entail, toStore and interpret.

Clearly, we have $d \in \mathscr{R}_{\Pi'}[\![A' \wedge B]\!]\, c$, where $d = \lim t \sqcup c$.

Suppose $d \in \mathscr{R}_{\Pi'}[\![A \wedge B]\!]\, c$. There are corresponding traces $t_1 \in \mathscr{O}_{\Pi'}[\![A']\!]$ and $t_2 \in \mathscr{O}_{\Pi'}[\![B]\!]$ such that $\lim t_1 = \lim t_2 = d$ and $\text{fn}(t_1) \cap \text{fn}(t_2) \cap (\bot \to c) = (\bot \to d)$. Let $(e_i)_{i\in\omega} = v(t_1) = v(t_2)$. If we consider traces t_1' and t_2' where $v(t_1') = v(t_2') = (\exists_L e_i)_{i\in\omega}$ and $r(t_1') = r(t_1')$ and $r(t_2') = r(t_2')$ it follows that $t_1' \in \mathscr{O}_{\Pi'}[\![A']\!]$ (this is easy to establish from the computation rules) and that $\text{fn}(t_2') \supseteq \text{fn}(\bar{t})$. Thus, $\text{fn}(t_1') \subseteq \text{fn}(t)$, so t must be a subtrace of t_1'. This implies that $t \in \mathscr{A}_{\Pi'}[\![A']\!]$, which contradicts the assumption that $t \notin \mathscr{A}_\Pi[\![A']\!]$.

10. ALGEBRAIC PROPERTIES OF ccp

One of the reasons for considering fully abstract semantics of a programming language is that since a fully abstract semantics is in a sense minimal, it follows that any algebraic equality that is satisfied by any (correct) semantics, will be satisfied by the fully abstract semantics. Thus, it is natural that we use the fully abstract semantics as a starting-point for the investigation of the algebraic properties of ccp. It turns out that the algebra of ccp agents satisfies the axioms of intuitionistic linear algebra (see Troelstra [36, chapter 8] and Ono [27]), which suggests a relationship between ccp and intuitionistic linear logic.

The work presented in this section is influenced by the results of Mendler, Panangaden, Scott and Seely [24], who show that a semantic model of ccp forms a hyperdoctrine [21, 34], a category-theoretic structure which represents the proof-theoretic structure of logics. Thus, proving that ccp is a hyperdoctrine implies that ccp in fact forms a logic. The authors of reference [24] stress the point; ccp *is* logic.

Other attempts to relate algebraic rules and concurrency include the work by Bergstra and Klop [3], in which algebraic rules were used to define a concurrent language, and Winskel and Nielsen [37], who relate different models of concurrency by examining their category-theoretic properties. Abramsky and Vickers [1] propose the use of quantales as a framework for the study of various aspects of concurrency. (The algebra of quantales is closely related to intuitionistic linear algebra.)

We begin by giving the axiomatic definition of intuitionistic linear algebra, following Troelstra [36]. Intuitionistic linear algebra is to linear logic as Boolean algebra is to propositional logic, i.e., an algebraic formulation of the derivation rules of the logic.

DEFINITION 10.1. An *IL-algebra* (intuitionistic linear algebra) is a structure $(X, \sqcup, \sqcap, \bot, \multimap, *, \mathbf{1})$ such that the following holds.

1. $(X, \sqcup, \sqcap, \bot)$ is a lattice.
2. $(X, *, \mathbf{1})$ is a commutative monoid.
3. If $x \leqslant x'$ and $y \leqslant y'$ it follows that $x * y \leqslant x' * y'$ and $x' \multimap y \leqslant x \multimap y'$.
4. $x * y \leqslant z$ iff $x \leqslant y \multimap z$.

In the definition, $*$ is to be seen as the multiplicative conjunction and $\sqcap$ as the additive conjunction.

Next we will see how the semantic domain of the fully abstract semantics can be seen as an IL-algebra. Let $\mathbf{A}$ consist of the subtrace-closed sets of traces. The lattice-structure of $\mathbf{A}$ is simply the inclusion-ordering of the sets of traces.

PROPOSITION 10.2. $\mathbf{A}$ *forms a complete distributive lattice.*

Proof. It is easy to see that for any family of subtrace-closed sets, the union and intersection of these sets is also subtrace-closed. From this follows also that $\mathbf{A}$ is a distributive lattice. ∎

Let $*\colon \mathbf{A} \times \mathbf{A} \to \mathbf{A}$ be parallel composition, i.e.,

$$x * y = \{t \mid t_1 \in x, t_2 \in y, \lim(t_1) = \lim(t_2), \mathrm{fn}(t) \supseteq \mathrm{fn}(t_1) \cap \mathrm{fn}(t_2)\}.$$

Let $\mathbf{1}$ be the set of passive traces, i.e., $\mathbf{1} = \{t \mid r(t) = \varnothing\}$. It is easy to see that $\mathbf{1}$ corresponds to the agent **true**, i.e. the tell constraint which always holds.

PROPOSITION 10.3. *$(\mathbf{A}, *, \mathbf{1})$ is a commutative monoid. For x and $\{y_i\}_{i \in I} \in \mathbf{A}$ the distributive law $x * (\bigcup_{i \in I} y_i) = \bigcup_{i \in I} x * y_i$ holds.*

Define $\multimap : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$ according to $x \multimap y = \bigcup \{z \mid x * z \subseteq y\}$. It follows that $x \multimap \cdot$ is an upper adjoint of $\cdot * x$, i.e., that $x \subseteq y \multimap z$ if and only if $x * y \subseteq z$ holds for $x, y, z \in \mathbf{A}$.

We also define an upper adjoint to $\cap$, even though this is not necessary to satisfy the axioms of IL-algebras. Let $\succcurlyeq : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$ according to $x \succcurlyeq y = \bigcup \{z \mid x \cap z \subseteq y\}$ gives us $x \subseteq y \succcurlyeq z$ if and only if $x \cap y \subseteq z$, for $x, y, z \in \mathbf{A}$.

It follows immediately that the structure we have obtained satisfies the axioms of IL-algebras.

THEOREM 10.4. *$(\mathbf{A}, \cup, \cap, \varnothing, \multimap, *, \mathbf{1})$ as defined above is an IL-algebra.*

Next, we will take a look at how selection in ccp can be expressed using the operations of IL-algebra.

First, note that the functions $\multimap$ and $\succcurlyeq$ can be expressed directly in terms of sets of traces. For traces t_1, t_2, let $t_1 \vee t_2$ be defined when $v(t_1) = v(t_2)$, and $u = t_1 \vee t_2$ be such that $v(u) = v(t_1)$, and $r(u) = r(t_1) \cup r(t_2)$. We find that

$$x \multimap y = \{t \mid \text{if } u \in X \text{ and } t \vee u \text{ is defined, we have } t \vee y \in y\}.$$

If there were no restriction that the elements of $\mathbf{A}$ must be subtrace-closed, $x \succcurlyeq y$ would consist of the traces which do not belong to x, together with the traces of y, similar to the usual definition of implication in classical logic. But since the complement of an element of $\mathbf{A}$ in general is not subtrace-closed, the traces of $x \succcurlyeq y$ are instead given by

$$x \succcurlyeq y = \{t \mid \text{if } u \in x \text{ is a subtrace of } t, \text{ then } u \in y\}.$$

For $x \in \mathbf{A}$, let the negation $\sim x$ be given as $\sim x = x \multimap \varnothing$. The negation of x can also be given directly as a set of traces according to

$$\sim x = \{t \mid \text{there is no } u \in x \text{ such that } \lim(t) = \lim(u)\}.$$

For an agent which is a tell constraint c, the set of traces is

$$c = \{t \mid \lim(t) \sqsupseteq c; \mathrm{fn}(t) \sqsubseteq (\bot \to c)\},$$

writing c for the set of traces of the tell constraint c. Also, note that

$$c \gg \mathbf{1} = \{t \mid \mathrm{fn}(t) \cup (\bot \to c) = \mathscr{U}\}.$$

For a tell constraint c, $\sim\sim c$ is the set of traces with limit at least c.

For $a \in \mathbf{A}$, the expression

$$a \cap (c \gg \mathbf{1}) \cap \sim\sim c$$

gives the set of traces t of a which satisfy $\lim t \sqsupseteq c$ and $\mathrm{fn}\, t = (c \to \mathrm{fn}(t))$, i.e., the set of traces corresponding to the alternative $c \Rightarrow A$ in a selection.

Given constraints c_1 and c_2, the expression

$$\sim c_1 \cap \sim c_2 \cap 1$$

corresponds to the set of traces in which neither c_1 nor c_2 ever become entailed by the store.

The set of traces of a selection $(c_1 \Rightarrow A_1 \,\square\, c_2 \Rightarrow A_2)$ can thus be given by the expression

$$(a_1 \cap (c_1 \gg \mathbf{1}) \cap \sim\sim c_1) \cup (a_2 \cap (c_2 \gg \mathbf{1}) \cap \sim\sim c_2) \cup (\sim c_1 \cap \sim c_2 \cap \mathbf{1}),$$

where a_k is the set of traces given by the abstract semantics of A_k, for $k \in \{1, 2\}$. (This translation can easily be generalised to selections with an arbitrary number of alternatives.) So non-deterministic selection can be defined using operations derived from parallel composition and the inclusion-ordering of sets of traces.

11. CONCLUSIONS

One reason to consider fully abstract semantics is that the set of algebraic identities satisfied by a fully abstract semantics will be the identities satisfied by any semantics. In the case of ccp, the fully abstract semantics turns out to satisfy the axioms of intuitionistic linear algebra, an algebra which was defined to capture the properties of intuitiorlistic linear logic. It is also interesting to note that selection can be expressed using other operations of intuitionistic linear logic. It is difficult to judge the importance of these results, but the match between ccp and intuitionistic linear algebra seems too strong to be dismissed as a coincidence.

Two proofs of full abstraction were given. The first relied on the use of infinite conjunctions to provide an appropriate context and as it could be argued that this context is not a realistic program we gave a second proof in which the context was finite but depended on an infinite input. It is worthwhile to ask whether it really is necessary to introduce infinite information in the context. After all, the set of finite

agents is countable (if we make some reasonable assumptions on the constraint system) so one would expect that a countable set of contexts should be sufficient to distinguish agents with differing behaviour. Even though the set of traces of an agent are in general uncountable it may be possible to select a set of "computable" traces so that if two agents differ in behaviour, there is some computable trace that one agent can exhibit but not the other. What is interesting here is not (only) the prospect of finding a slightly more general proof of full abstraction, but also the idea that a there may be a countable set of traces that can capture the infinite behaviour of agents.

One issue that is *not* addressed in this paper is the problem of giving a fully abstract *fixpoint* semantics for ccp. Boudol [4] showed that it was not possible to give a continuous fully abstract fixpoint semantics for a language which allows non-determinism and arbitrary recursion, if one wants to consider the results of infinite computations. Park [29] showed that for a non-deterministic programming language with a fairness property it was possible to write a program that exhibits a form of unbounded non-determinism (see also Example 3.5). It is thus possible to apply a related result by Apt and Plotkin [2] where it is shown that in a language with unbounded non-determinism it is not possible to give a fully abstract least fixed point semantics. The first author of the current paper has presented a more general result, which shows that even if we are willing to give up continuity it is still not possible to construct a fully abstract semantics for a wide class of non-deterministic programming languages, provided that we want the semantics to allow infinite observations (see [25, Chapter 7] and [26]).

As it is not possible to define a fully abstract fixpoint semantics for a concurrent language like ccp, are there any other criteria one could use in the evaluation of a fixpoint semantics? One would like the semantics to be simple and to preserve as many algebraic properties of the fully abstract semantics as possible. The semantics should of course not be a mere reflection of the syntactic form of programs, but present the meaning of a program in a form which is close to the fully abstract semantics. In [25, Chapter 9], the first author presents a fixpoint semantics for ccp developed along these criteria.

Received January 18, 1995; final manuscript received March 18, 1998.

REFERENCES

1. Abramsky, S., and Vickers, S. (1990), Quantales, observational logic, and process semantics, Technical Report DOC 90/1, Imperial College, Dept. of Computing, January.
2. Apt, K. R., and Plotkin, G. D. (1986), Countable nondeterminism and random assignment, *J. ACM* **33**(4), 724–767.
3. Bergstra, J. A., and Klop, J. W. (1984), Process algebra for synchronous communication, *Inform. and Control* **60**, 109–137.
4. Boudol, G. (1981), Une sémantique pour les arbres non déterministes, *in* "Proceedings of the 6th Colloquium on Trees in Algebra and Programming (CAAP '81)" (E. Astesiano and C. Böhm, Eds.), LNCS, Vol. 112, pp. 147–161.

5. Brock, J. D., and Ackerman, W. B. (1981), Scenarios: A model of non-determinate computation, *in* "Formalization of Programming Concepts" (Diaz and Ramas, Eds.), LNCS, Vol. 107, pp. 252–259.
6. Brooks, S. D., Hoare, C. A. R., and Roscoe, A. W. (1984), A theory of communicating sequential processes, *J. ACM* **31**(3), 560–599.
4. Brookes, S. (1993), Full abstraction for a shared variable parallel language, *in* "Proc. 8th IEEE Int. Symp. on Logic in Computer Science," 98–109.
8. Carlson, B. (1991), "An Approximation Theory for Constraint Logic Programs," thesis for the Degree of Licientiate of Philosophy, Uppsala University.
9. de Boer, F. S., Kok, J. N., Palamidessi, C., and Rutten, J. J. M. M. (1991), The failure of failures in a paradigm for asynchronous communication, *in* "Proceedings of CONCUR '91," LNCS, Vol. 527, pp. 111–126, Springer-Verlag, Berlin/New York.
10. de Boer, F. S., and Palamidessi, C. (1991), A fully abstract model for concurrent constraint programming, *in* "TAPSOFT" LNCS, Vol. 493, pp. 296–319.
11. de Boer, F. S., Di Pierro, A., and Palamidessi, C. (1995), Nondeterminism and infinite computations in constraint programming, *Theoret. Comput. Sci.* **151**, 36–78.
12. Foster, I., and Taylor, S. (1989), "Strand: New Concepts in Parallel Programming," Prentice–Hall, New York.
13. Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislowe, M., and Scott, D. S. (1980), "A Compendium of Continuous Lattices," Springer-Verlag, Berlin/New York.
14. Henkin, L., Monk, J. D., and Tarski, A. (1971), "Cylindric Algegras," Vol. 1, North-Holland, Amsterdam.
15. Jagadeesan, R., Pingali, K., and Panangaden, P. (1991), A fully abstract semantics for a functional programming language with logic variables, *TOPLAS* **13**(4), 577–625.
16. Jagadeesan, R., Saraswat, V. A., and Shanbhogue, V. (1991), Angelic nondeterminism in concurrent constraint programming, technical report, System Sciences Laboratory, Xerox PARC, January.
17. Jonsson, B. (1994), A fully abstract trace model for dataflow and asynchronous networks, *Distrib. Comput.* **7**, 197–212.
18. Kahn, G. (1974), The semantics of a simple language for parallel programming, *in* "Proceedings of IFIP Congress," pp. 471–475, North-Holland, Amsterdam.
19. Keller, R. M. (1978), Denotational models for parallel programs with indeterminate operators, *in* "Formal Descriptions of Programming Concepts" (Neuhold, Ed.), pp. 337–366, North-Holland, Amsterdam.
20. Kwiatkowska, M. (1992), Infinite behaviour and fairness in concurrent constraint programming, *in* "Semantics: Foundations and Applications," LNCS, Vol. 666, pp. 348–383, Springer-Verlag, Berlin/New York.
21. Lawvere, F. W. (1969), Adjointness in foundations, *Dialectica* **23**(3/4), 281–296.
22. Maher, M. J. (1987), Logic semantics for a class of committed-choice programs, *in* "4th International Conference on Logic Programming," pp. 858–876, MIT Press, Cambridge, MA.
23. McCarthy, J. (1967), A basis for a mathematical theory of computation, *in* "Computer Programming and Formal Systems" (P. Brafford and D. Hirschberg, Eds.), pp. 33–70, North-Holland, Amsterdam.
24. Mendler, N. P., Panangaden, P., Scott, P. J., and Seely, R. A. G. (1995), A logical view of concurrent constraint programming, *Nordic J. Comput.* **2**(2), 181–220.
25. Nyström, S. (1996), "Denotational Semantics for Asynchronous Concurrent Languages," Ph.D. thesis, Uppsala University.
26. Nyström, S. (1996), There is no fully abstract fixpoint semantics for nondeterministic languages with infinite computations, *Inform. Process. Lett.* **60**(6), 289–293.
27. Ono, H. (1990), Phase structures and quantales—A semantical study of logics without structural rules, Lecture delivered at the conference, *in* "Logics with restricted structural rules," University of Tübingen, October. [Cited in [36]]

28. Palmgren, E. (1994), Denotational semantics of constraint logic programs–A non-standard approach, *in* "Constraint Programming" (B. Mayoh, E. Tyugu, and J. Penjam, Eds.), NATO ASI Series F, pp. 261–288, Springer-Verlag, Berlin/New York.
29. Park, D. (1980), On the semantics of fair parallelism, *in* "Abstract Software Specifications, Copenhagen Winter School 1979," LNCS, Vol. 86, pp. 504–526, Springer-Verlag, Berlin/New York.
30. Plotkin, D. (1981), "A structural Approach to Operational Semantics," Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark.
31. Russell, J. R. (1989), Full abstraction for nondeterministic dataflow networks, *in* "Proc. 30th Annual Symp. Foundations of Computer Science," pp. 170–177.
32. Saraswat, V. A., Rinard, M., and Panangaden, P., (1991), Semantic foundations of concurrent constraint programming, *in* "Proc. 18th ACM Symp. on Principles of Programming Languages."
33. Scott, D. S. (1982), Domains for denotational semantics, *in* "ICALP '82," LNCS, No. 140, pp. 577–613, Springer-Verlag, Berlin/New York.
34. Seely, R. A. G. (1983), Hyperdoctrines, natural deduction and the Beck condition, *Z. Math. Logik Grundlagen Math.* **29**, 505–542.
35. Shapiro, E. Y. (1983), "A Subset of Concurrent Prolog and Its Interpreter," Technical Report 003, Institute for New Generation Computer Technology, Tokyo.
36. Troelstra, A. S. (1992), "Lectures on Linear Logic," CSLI Lecture Notes, No. 29, Center of the Study of Language and Information, Stanford.
37. Winskel, G., and Nielsen, M. (1993), "Models for Concurrency," Technical Report PB-463, Computer Science Department, Aarhus University. [To appear in "Handbook of Logic in Computer Science," Oxford Univ. Press, London]