UPMAIL Technical Report No. 79
14 July, 1994

Indeterminate Concurrent Constraint Programming: A Fixpoint Semantics for Non-Terminating Computations

Sven-Olof Nyström
Computing Science Department,
Uppsala University
P. O. Box 311, S-751 05 Uppsala, Sweden
svenolof@csd.uu.se

Bengt Jonsson
Department of Computer Systems,
Uppsala University
P. O. Box 311, S-751 05 Uppsala, Sweden
bengt@docs.uu.se

Abstract

This paper presents a semantics for non-deterministic concurrent constraint programming languages. The semantics address the issues of giving an adequate treatment of non-terminating computations, in particular with fairness and liveness, and giving a compositional semantics, where the meaning of a recursively defined program is obtained as a fixpoint. We present a simple concurrent constraint programming language with a reduction rule semantics, which is augmented with fairness requirements. We specify how traces are obtained from fair computations of a program. The operational semantics of a program is defined as the set of traces of the program. We show that this trace semantics can be defined compositionally. The trace semantics is transformed into a continuous fixpoint semantics, using the framework of Lehmann's category-theoretic constructions, and present a general schema for defining continuous operators on the semantic domain. A simple abstraction operator is defined, and is proved to give a fully abstract semantics.

This article was published in the proceedings of the 1993 international symposium on logic programming.

1 Introduction

The concurrent constraint programming languages [7, 10] provide a simple and powerful model of asynchronous concurrent computation. The main feature of constraint programming 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. The main computation 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 purpose of this paper is to present a denotational semantics for nondeterministic concurrent constraint programming languages, which in particular can address the issues of

1. giving an adequate treatment of non-terminating computations, and in particular of fairness and liveness,
2. giving a *fully abstract* semantics, and
3. giving a compositional semantics, where the meaning of a recursively defined program is obtained as a fixpoint.

Note that previous works have not treated nondeterministic and non-terminating computations.

It has now become rather standard to describe non-deterministic programs by the set of traces of the computations of the program. However, to obtain a fixpoint semantics for recursive programs, it is necessary to find a suitable domain in which fixpoints of functions over sets can be computed. Previous research on fixpoint semantics has provided methods to handle either non-terminating behavior of deterministic programs, or finite behaviors of nondeterministic programs. It is difficult to handle both non-terminating behavior and nondeterminism, in particular in the presence of fairness. For instance, Apt and Plotkin [2] have shown that it is impossible to have a continuous, least fixed-point, fully abstract semantics using domain theory.

Earlier works [3, 10] have shown that the operational semantics of concurrent constraint programming languages can be formulated rather easily using structural operational semantics. For example, Boer and Palamidessi [3] have presented a fully abstract structural operational semantics for non-recursive programs, which descibes finite behavior. For a deterministic concurrent constraint language Saraswat, Rinard, and Panangaden [10] have presented a semantics quite similar to Kahn's semantics [4], which handles finite and infinite computations in a uniform manner. They also generalize this semantics to a non-deterministic language. However, this semantics can only describe finite behaviors, and is thus unable to capture aspects pertaining to fairness or liveness.

In this paper, we will adopt category theoretic methods developed by Lehmann [5, 6]. A category can be seen as a generalization of a cpo, since

any cpo is also a category. The advantage of using categories lies in the fact that they can convey more information about the described programs, and that this extra information can be used to obtain continuous operators and fixpoints, even in the presence of fairness. Lehmann's ideas have previously been used by Abramsky [1] to give a semantics for an applicative language and by Panangaden and Russell [8] for a simple imperative sequential language with unbounded nondeterminism. In this paper, we apply Lehmann's ideas to a trace-based semantics for a concurrent language.

The contents of this paper are the following.

We present a simple concurrent constraint programming language similar to that presented in [10]. A reduction rule semantics of the language is given using structurally defined reduction rules. The semantics is augmented with fairness requirements. We specify how traces are obtained from fair computations of a program. The operational semantics of a program is defined as the set of traces of the program. We show that this trace semantics can be defined compositionally.

In a next part of the paper, we transform the previous trace semantics into a continuous fixpoint semantics. We present the framework of Lehmann's category-theoretic constructions, and present a general schema for defining continuous operators on the semantic domain. This general schema appears to be very general and is of independent interest. We apply the schema to present continuous operators corresponding to the operators on sets of traces given earlier.

The fixpoint semantics is compositional, but it is not fully abstract with respect to the simplest form of semantics, that which only records the final value of a computation, or the limit of an infinite computation. In the final part of the paper, we present a semantics which is also fully abstract. We obtain this semantics via a closure operation from the fixpoint semantics. The fully abstract semantics can be understood also independently of the context of a fixpoint semantics.

2 Constraint Systems

In this section, we give a brief presentation of constraint systems that are used in the remainder of the paper. The exact form of the constraint system is not so important for our results. We consider here a rather simple form of constraint system based on a first order language.

So, assume a first-order language containing the usual connectives, variables, predicate symbols, function symbols and constant symbols. We will further assume that there is an interpretation I of the constant, function, and predicate symbols into some domain C. Let Var be the set of variables, and Val the set of assignments of variables $V : Var \to C$. We will typically use the letters x and y for variables, and X and Y for finite sequences of variables with no repetitions. For a formula ϕ, write $\mathrm{FV}(\phi)$ for the set of free variables of ϕ. We write $V \models_I \phi$ if V is an assignment of variables such

that ϕ is true in the interpretation I.

Example 1 Let C be the set of natural numbers, let the language have one binary predicate symbol $\leq$, no function symbols, and a constant symbol for each natural number. The definition of $\models$ for this constraint system should be obvious.

For a formula ϕ and sequences of variables X and Y of equal length, we define $\phi[Y/X]$ as a formula. We extend the definition of $\models_I$ so that $V \models_I \phi[Y/X]$ iff $V' \models_I \phi$, where V' is the assignment such that $V(y_i) = V'(x_i)$, for $i \leq |X|$, and $V(x) = V'(x)$ for $x \in \mathit{Var} \setminus X$. The reader is invited to check that the properties of an expression $\phi[Y/X]$ are exactly those one would expect of a reasonable definition of syntactic substitution.

We define a preorder $\preceq$ between formulas by $\phi \preceq \psi$ iff for any $V \in \mathit{Val}$ such that $V \models_I \psi$, we have $V \models_I \phi$. This gives immediately an equivalence relation $\phi \equiv \psi$ iff $\phi \preceq \psi$ and $\psi \preceq \phi$.

Definition 2 A *constraint* is a non-empty set c of formulas, such that

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

For a formula ϕ let $[\phi] = \{\psi \mid \psi \preceq \phi\}$. Clearly $[\phi]$ is a constraint. A constraint c for which there is an formula ϕ such that $[\phi] = c$ is a *finite constraint*. If we have constraints $c_0, c_1, c_2, c_3, \ldots$ such that $c_0 \subseteq c_1 \subseteq c_2 \subseteq c_3 \ldots$ then $\cup_{i \geq 0} c_i$ is a constraint.

Let D be the set of constraints, and D_c the set of finite constraints.

We will use $I_Y^X(\phi)$ as an abbreviation for $(\exists x_1 \ldots \exists x_n . \phi)[Y/X]$, where $\{x_1, \ldots, x_n\} = (\mathrm{FV}(\phi) \setminus X)$. This construct will be used to handle procedure calls with parameter passing and local variables. So I_Y^X hides all variables that do not occur in X, and renames the ones mentioned in X to the corresponding ones in Y.

For example, given the constraint system above, $I_y^x(x \leq 2, z \leq 3) = (y \leq 2)$.

We extend conjunction and renaming to be defined for finite constraints, so for formulas ϕ and ψ, $[\phi] \wedge [\psi] = [\phi \wedge \psi]$ and $I_Y^X([\phi]) = [I_Y^X(\phi)]$. It is straight-forward to prove that these operations are well-defined. Also, note that they are monotone with respect to $\sqsubseteq$, and for finite c, d, the constraints $I_Y^X(c)$, and $c \sqcup d$ will also be finite. In the following text, we will use the letters c, d ... for elements of D.

Note that each constraint is either finite, or a limit of a directed set of finite constraints. So the constraints form a Scott domain (and a cpo), and we will use the usual relation symbol $\sqsubseteq$, so that $c \sqsubseteq d$ if $c \subseteq d$, and the symbol $\sqcup$ for least upper bound, which happens to coincide with conjunction.

In the following text, we will not refer to the details of the construction of the constraints, but only refer to D_c, D, the function I_Y^X (and the properties of this function), and the other operations which come with cpo's.

3 The language

We will describe a concurrent constraint language, similar to that presented by Saraswat [10]. We assume a set NAME of procedure symbols $p, q, \ldots$. The syntax of an *agent* A is given as follows.

$$A ::= c \mid \bigwedge_{i \in S} A_i \mid p(Y) \mid (c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n) \mid A \langle c \rangle (Y/X) \quad .$$

A tell constraint, written c, is assumed to be a member of D_c. The conjunction

$$\bigwedge_{i \in S} A_i$$

of agents, where S is assumed to be countable, represents a parallel composition of the agents A_i. We will use $A_1 \wedge A_2$ as a shorthand for $\bigwedge_{i \in \{1,2\}} A_i$.

Agents of the form $A \langle c \rangle (Y/X)$ are called *closures*. They occur when a call to a procedure has been replaced with the body of its definition. X is the sequence of formal parameters, Y the sequence of actual ones, and the constraint c represents the local data. An agent of the form $p(Y)$ represents a call to a procedure in the program.

An agent $(c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n)$ represents a selection. If one of the ask constraints c_i becomes true, the corresponding agent A_i may be executed.

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. Note that the syntax for agents describe both agents appearing in a program, and agents appearing as intermediate states in a computation. Agents appearing on the left-hand side of a definition will actually have a form indicated by the following syntax.

$$A ::= c \mid A \wedge A \mid p(X) \mid c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n$$

The differences are that in a program, all conjunctions are finite, and that closures do not occur.

4 Configurations and Computation Rules

A *configuration* is a pair $A : c$ consisting of an agent A acting on a constraint c. The latter will be referred to as the *environment* of the state. The operational semantics is given through a relation $\longrightarrow$ over configurations, assuming a program Π. Intuitively, a computation step $A : c \longrightarrow A' : c'$ denotes that the configuration $A : c$ can be reduced to $A' : c'$ in one step. c' will always contain more information than c ($c \sqsubseteq c'$) so a computation step is never destructive. The result of a finite computation would be the final environment. What about infinite computations? Here, the result should be the least upper bound of all environments in the computation.

We will present rules that define $\longrightarrow$ in the usual style of structural operational semantics. A rule typically has the form

$$\frac{P_1, \ldots, P_n}{A : c \longrightarrow A' : c'} \ ,$$

and should be read: "The configuration $A : c$ can be reduced to $A' : c'$ whenever the premises $P_1, \ldots, P_n$ hold."

An agent which is a *tell constraint* simply adds new information (itself) to the environment.

$$\frac{}{c : d \longrightarrow c : c \sqcup d}$$

A *conjunction* of agents is executed by interleaving the execution of the agents. One computation step by the conjunction involves one step by exactly one of the agents.

$$\frac{A_k : c \longrightarrow A' : d, \quad k \in S}{\bigwedge_{i \in S} A_i : c \longrightarrow \bigwedge_{i \in S} B_i : d}$$

where $B_i = A_i$, for $i \in S \setminus \{k\}$, and $B_k = A'$.

A *call* to a procedure is reduced to the body of its definition, wrapped in a closure. Note that we assume that each call has the same number of parameters as the corresponding definition.

$$\frac{p(X) :: A \quad \in \ \Pi}{p(Y) : c \longrightarrow A \, \langle \bot \rangle \, (Y/X) : c}$$

As mentioned before, closures provide renaming of parameters and hiding of local data. A closure $A \, \langle c \rangle \, (Y/X)$ is executed one step by doing the following. Apply the function I_X^Y to the present environment, denoted d say, to extract the actual parameters Y and replace them by the formal parameters X. Combine the result $I_X^Y(d)$ with the local data (given by c), to obtain a local environment. A computation step is performed in the local environment, which gives a new local environment (e say). The local environment e is saved in the closure. To transmit any results to the global environment, the function I_Y^X is applied to the local environment to hide local variables and rename formal parameters to actual parameters. The constraint thus obtained is combined with the previous global environment. The rule thus becomes

$$\frac{A : c \sqcup I_X^Y(d) \longrightarrow A' : e}{A \, \langle c \rangle \, (Y/X) : d \longrightarrow A' \, \langle e \rangle \, (Y/X) : I_Y^X(e) \sqcup d}$$

The reduction rule for selection is the following. If one of the ask constraints is satisfied in the current environment, the selection can be reduced to the corresponding agent. If several ask constraints are satisfied, any of the alternatives may be selected.

$$\frac{c_i \sqsubseteq c}{(c_1 \rightarrow A_1 \ [] \ \ldots \ [] \ c_n \rightarrow A_n) : c \longrightarrow A_i : c}$$

5 Computations and Fairness

A computation is an infinite sequence of configurations, in which each adjacent pair of configurations is either a computation step (performed by the agent), or a strengthening of the environment (performed by some other agent). Throughout this chapter, we assume a fixed program Π.

Definition 3 A *computation* is a sequence of configurations $(A_i : c_i)_{i\in\omega}$ such that for all $i \geq 0$, we have either $A_i : c_i \longrightarrow A_{i+1} : c_{i+1}$ (*a computation step*), or $A_i = A_{i+1}$ and $c_i \sqsubseteq c_{i+1}$ (*an input step*).

If a computation could be finite, when the result of a finite computation would be the final environment. What about infinite computations? Here, the result should be the least upper bound of all environments in the computation. So a computation may produce its result by performing computation steps and by receiving input. A computation that does not receive any input is called a *non-interactive computation*. An input step which does not alter the environment is called an empty input step. We shall require computations to be fair. This should intuitively mean that all computation steps that can performed will eventually be performed. For instance, the natural requirement for selection statements is that if one selection can be performed, then either that reduction, or any other selection, should eventually be performed. A call to a procedure, which occurs in a computation, should eventually be reduced to its definition.

Before we can turn to the actual definition of fairness, we must define some auxiliary properties of computations. 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 4 The relation '*immediate inner computation of*' is defined to be the least relation over ω-sequences of processes such that the following holds.

1. A sequence $(A_i^j : c_i)_{i\in\omega}$ is an immediate inner computation of $(\bigwedge_{j\in S} A_i^j : c_i)_{i\in\omega}$, if $j \in S$.
2. The sequence $(A_i : c_i \sqcup I_X^Y(d_i))_{i\in\omega}$ is an immediate inner computation of the sequence $(A_i \langle c_i \rangle (X/Y) : d_i)_{i\in\omega}$.

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

Proposition 5 If $(A_i : c_i)_{i\in\omega}$ is an inner computation of $(B_i : d_i)_{i\in\omega}$, and $(B_i : d_i)_{i\in\omega}$ is a computation, then $(A_i : c_i)_{i\in\omega}$ is a computation.

For a sequence $(x_i)_{i\in\omega}$ and $k \geq 0$, we define $((x_i)_{i\in\omega}) \uparrow k$ as the sequence $(x_{i+k})_{i\in\omega}$.

Lemma 6 Let x be a computation with inner computation y. For an arbitrary $k \in \omega$, $y \uparrow k$ is an inner computation of $x \uparrow k$.

We are now ready to consider the actual definition of fairness. We will define the fairness requirement in a bottom-up fashion by giving a sequence of fairness requirements of increasing strength. First, we define a class of computations where the requirement is that if the statement part of the first process is a selection where one of the tests evaluates to true, it should be reduced to one of its branches eventually, a call to a program defined procedure should be reduced to its definition eventually, and a call to a primitive procedure should produce the desired result eventually.

Definition 7 A computation $(A_i : c_i)_{i\in\omega}$ is *top-level fair* when the following holds.

1. If $A_0 = p(Y)$, and the program contains a definition $p(X) :: A \in P$, there is an $i \geq 0$ such that $A_i = A(Y/X)$.
2. If $A_0 = c$, there is an $i \geq 0$ such that $c_i \sqsupseteq c$.
3. If $A_0 = (d_1 \Rightarrow B_1 \,[]\, d_2 \Rightarrow B_2 \,[]\, \ldots \,[]\, d_n \Rightarrow B_n)$, and $d_j \sqsubseteq c_0$ for a $j \leq n$, then there is an $i \geq 0$ such that $A_i{=}B_j$ for a $j \leq n$.

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

6 Operational Semantics

In this section, we first present the most abstract semantics (the result semantics) which simply represents an agent by the possible results of its computations, when performed in isolation. We then present a more refined semantics (the trace semantics) which represents the computations of an agent, but abstracts from the local state of the agent. The result semantics is a function $\mathcal{R}(\text{NAME} \to \text{STAT}) \to \text{STAT} \to D_c \to \wp(D)$ which gives the set of all possible results that can be computed given a program, an agent and an initial environment.

$$\mathcal{R}_\Pi[\![A]\!]c = \{\sqcup_{i\in\omega} c_i \mid (A_i : c_i)_{i\in\omega} \text{ is a fair non-interactive computation where } c_0 = c \text{ and } A_0 = A\}.$$

Remember that a computation is defined to be a sequence of configurations $(A_i : x_i)_{i\in\omega}$, where the environments, that is, the x_i's, are the only part of the computation visible to the outside. Now, a computation can go from $A_i : x_i$ to $A_{i+1} : x_{i+1}$ either by performing a computation step, or by receiving input, and this distinction is of course relevant to the semantics of a process.

In the trace semantics, a process is represented by a set of traces, where each trace is an infinite sequence of environments together with information which identifies computation steps.

Definition 8 A *trace* t is a pair $t = (v(t), r(t))$, where $v(t)$ is an ω-chain in D_c and $r(t) \subseteq \omega$. The set of traces is denoted TRACE.

The trace of a computation $(A_i : c_i)_{x \in \omega}$ is $t = ((c_i)_{i \in \omega}, r)$, where the ith step is a computation step when $i \in r$, and an input step, when $i \notin r$. For $t \in$ TRACE, $v(t)$ will sometimes be referred to as the *environment* of the trace. We will sometimes use the notation $v(t)_i$ to refer to the ith element of the environment of t.

A trace t where $r(t) = \emptyset$ can be characterized as passive, since this trace corresponds to a computation with input actions only. Similarly, if we have traces t and u such that $v(t) = v(u)$ and $r(t) \subseteq r(u)$, t can be viewed as more passive than u, or, conversely, one can say that u is the more active of the two. Therefore it is natural to define an ordering between traces.

Define the relation $\leq \subseteq$ TRACE $\times$ TRACE such that $t \leq u$ iff $v(t) = v(u)$ and $r(t) \subseteq r(u)$. This is obviously a partial order. Under this ordering, TRACE does not have a least element, but all non-empty, consistent sets $S \subseteq$ TRACE have least upper bounds. That is, S is consistent iff $v(t) = v(u)$ for all $t, u \in S$. The least upper bound $\bigvee S$ of the traces in S then becomes $(v(t), \cup_{u \in S} r(u)$ where t is an arbitrary element of S.

A trace t is maximal with respect to the order $\leq$ if there is no trace t' such that $t' > t$, that is if $r(t) = \omega$. A trace t is minimal if $r(t) = \emptyset$. In other words, maximal traces correspond to non-interactive computations with no input steps. However, since a non-interactive computation may contain empty input steps, any trace t such that for all $i \in \omega$, either $i \in r(t)$ or $v(t)_i = v(t)_{i+1}$ also corresponds to a non-interactive computation.

The operational semantics of an agent A, assuming a set of possibly recursive definitions (a program) Π is defined as follows.

Definition 9 The function $\mathcal{O} : (\text{NAME} \to A) \to A \to \wp(\text{TRACE})$ is defined so that $t = ((x_i)_{i \in \omega}, r) \in \mathcal{O}_\Pi[\![A]\!]$, iff there is a fair computation $(A_i : c_i)_{i \in \omega}$ in the program Π such that the ith step of $(A_i : c_i)_{i \in \omega}$ is a computation step when $i \in r$, and an input step, when $i \notin r$.

As one would expect, the result semantics can be obtained from the operational semantics.

Lemma 10 $d \in \mathcal{R}_\Pi[\![A]\!]c$ if and only if there is a $t \in \mathcal{O}_\Pi[\![A]\!]$ such that $v(t)_0 = c$, $d = \sqcup v(t)$, and for all $i \in (\omega \setminus r(t))$, $v(t)_i = v(t)_{i+1}$.

The operational semantics of a parallel composition $A \wedge B$ can be computed from the operational semantics of A and B. In the same way, the operational semantics of closures and selections can be computed from the operational semantics of their components. This is expressed by the following lemmas.

Lemma 11 $t \in \mathcal{O}_\Pi[\![\bigwedge_{i\in S} A_i]\!]$ iff there are $t_i \in \mathcal{O}_\Pi[\![A_i]\!]$ for $i \in S$, such that $v(t_i) = v(t)$ for $i \in S$, $r(t) = \bigcup_{i\in S} r(t_i)$, and $r(t_i) \cap r(t_j) = \emptyset$, for $i, j \in S$ such that $i \neq j$.

Lemma 12 $t \in \mathcal{O}_\Pi[\![A \langle c \rangle (X/Y)]\!]$ iff there is a $u \in \mathcal{O}_\Pi[\![A]\!]$ such that with $(d_i)_{i\in\omega} = v(t)$ and $(e_i)_{i\in\omega} = v(u)$, we have

1. $e_0 = c \sqcup I^Y_X(d_0)$,
2. for $i \in (\omega \setminus r(t))$, $e_{i+1} = e_i \sqcup I^Y_X(d_{i+1})$,
3. for $i \in r(t)$, $d_{i+1} = I^X_Y(e_{i+1}) \sqcup d_i$, and
4. $r(t) = r(u)$.

Lemma 13 $t \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow B_1 \,[]\, c_2 \Rightarrow B_2 \,[]\, \ldots \,[]\, c_n \Rightarrow B_n)]\!]$ iff one of the following holds.

1. $c_j \sqsubseteq v(t)_k$ for some $j \leq n$ and $k \geq 0$, and there is a $u \in \mathcal{O}_\Pi[\![B_j]\!]$ such that for all $i \geq 0$, $v(u)_i = v(t)_{i+k}$, $v(t)_k = v(t)_{k+1}$, and $r(t) = \{i + k + 1 \mid i \in r(u)\} \cup \{k\}$.
2. There is no $j \leq n$ and $k \geq 0$ such that $c_j \sqsubseteq v(t)_k$, and $r(t) = \emptyset$.

Proving these lemmas is fairly straight-forward using the computation rules and the fairness requirements. However, the composition rule for closures is in a form which is not suitable for the fixpoint semantics we will use, so we state the properties of the closure combinator in a slightly different form.

Lemma 14 $t \in \mathcal{O}_\Pi[\![A \langle c \rangle (X/Y)]\!]$ iff there is a trace $u \in \mathcal{O}_\Pi[\![A]\!]$ and a set R, where $r(u) \subseteq R \subseteq \omega$, such that with $(d_i)_{i\in\omega} = v(t)$ and $(e_i)_{i\in\omega} = v(u)$, the following holds.

1. $e_0 = c \sqcup I^Y_X(d_0)$.
2. For $i \in (\omega \setminus R)$, $e_{i+1} = e_i \sqcup I^Y_X(d_{i+1})$.
3. For $i \in R$, $d_{i+1} = I^X_Y(e_{i+1}) \sqcup d_i$.
4. $r(t) = \{i \in r(u) \mid d_{i+1} = d_i$, or
   there is no $j < i$ such that $j \in R \setminus r(u)\}$.

7 Powerdomains

In the search for domains suitable for the treatment of indeterminacy, Plotkin [9] and Smyth [12] introduced powerdomains as the domain-theoretic analogues to powersets. Plotkin examined and dismissed one very simple powerdomain construction; the representation of the powerdomain of the domain D as the function space $D \to \{\text{truth values}\}$. Under this construction, a set S_1 can only be approximated by a set S_2 if S_2 is a subset of S_1. This is not

satisfactory as one would like a set $\{x\}$ to be approximated by a set $\{y\}$ if y is an approximation of x. Plotkin and Smyth proposed various orderings but it does not seem possible to find a partial order for a powerdomain that has all desired properties.

Lehmann [5, 6] suggested a different approach. Instead of using partially ordered sets as domains, he proposed the use of (a class of) categories. Since any partially ordered set can be seen as a category, this is a rather straightforward generalization. For powerdomains, his solution takes advantage of the fact that two objects can have more than one arrow between them, and that thus an arrow can contain information about the way in which one object is an approximation of another.

Lehmann's construction has previously been used by Abramsky [1] and by Panangaden and Russell [8] to model unbounded indeterminacy.

Lehmann also showed how various domain-theoretic constructions (like sums, products, function spaces and least fixpoints of domain equations) had counterparts in category-theory. However, in this paper we will only consider powerdomains of partially ordered sets.

The following presentation of Lehmann's powerdomain construction follows the one in [8]. Given a complete partially ordered set $(B, \leq)$, the construction of the categorical powerdomain, $CP(B)$, is as follows. An object of $CP(B)$ is a multiset of elements of B (it is assumed that repeated elements of a multiset can somehow be distinguished). An arrow $G : a \to b$ of $CP(B)$ is a relation $G \subseteq a \times b$ such that

1. for x and y such that $\langle x, y\rangle \in G$, we have $x \sqsubseteq y$, and
2. for all $y \in b$, there is an $x \in a$ such that $\langle x, y\rangle \in G$.

The composition of two arrows $G_1 : a \to b$ and $G_2 : b \to c$ is as follows. $\langle x, z\rangle \in G_2 \circ G_1$ if there is a $y \in b$ such that $\langle x, y\rangle \in G_1$ and $\langle y, z\rangle \in G_2$. It should be clear that this composition fulfills the requirements on arrows. For an object a, the identity arrow $\mathbf{id}_a : a \to a$ is simply the relation $\mathbf{id}_a = \{\langle x, x\rangle \mid x \in a\}$. We leave it to the reader to verify that this arrow indeed satisfies the category-theoretic axioms for the identity arrow.

The category-theoretic analogue of an ω-sequence in a cpo is an ω-diagram, and limits of ω-sequences in cpo's have their counterparts in colimits of ω-sequences in categories. These are present in categories $CP(D)$ where D is such that consistent sets in D have least upper bounds.

For a diagram $a_0 \xrightarrow{G_0} a_1 \xrightarrow{G_1} a_2 \xrightarrow{G_2} \ldots$ the colimit has the following form. Let S be the set of all chains $v = (x_i)_{i\in\omega}$ such that for all $i \in \omega$, $\langle x_i, x_{i+1}\rangle \in G_i$. The colimiting object a^* is the set $\biguplus\{\sqcup v \mid v \in S\}$, and it forms the colimit together with arrows $H_j : a_j \to a^*$ such that $\langle x, y\rangle \in H_j$ iff there is a chain $v = (x_i)_{i\in\omega} \in S$ such that $x_j = x$ and $\sqcup v = y$.

Note that the colimiting object does not only depend on the objects $a_0, a_1, \ldots$, but also on the arrows between the objects.

Definition 15 A category where all diagrams $a_0 \xrightarrow{G_0} a_1 \xrightarrow{G_1} a_2 \xrightarrow{G_2} \ldots$ have colimits is an ω-category. A functor which preserves ω-colimits is an ω-functor.

An object a of a category is *initial* if for any object b there is exactly one arrow $G : a \to b$. This corresponds to the least element of a cpo. The initial object is often denoted by the symbol 0. If B has a least element $\perp$, the initial element of $CP(B)$ is $\{\perp\}$. If B does not have a least element, but there is a set $S \subseteq B$ such that each element of S is minimal, and for each member of B there is a weaker element in S, then S is initial in $CP(B)$.

Now, what does an ω-functor look like? It is a theorem of category theory that the ω-functors are closed under composition. Lehmann proved that (binary) disjoint union is an ω-functor.

In this section, we present a general class of ω-functors. The presentation will be in the form of a general schema, which can be used to define a large class of ω-functors.

Proposition 16 Let S be a set. For each $s \in S$, let $f_s : B \to B$ and $p_s : B \to \{\perp, \mathbf{true}\}$ be continuous functions. The functor which maps each object $a \in CP(B)$ to the object

$$F(a) = \biguplus_{s \in S} \{f_s(x) \mid p_s(x) \neq \mathbf{true} \text{ and } x \in a\}.$$

and maps each arrow $G : a \to b$ to

$$F(G) : F(a) \to F(b) = \biguplus_{s \in S} \{\langle f_s(x), f_s(y)\rangle \mid \langle x, y\rangle \in G,\ p_s(y) \neq \mathbf{true}\}.$$

is an ω-functor over $CP(B)$.

So a function $F : \wp(B) \to \wp(B)$ has a corresponding ω-continuous functor over $CP(B)$ if it can be written in the following form:

$$\begin{aligned} F(A) = \{f(a, x_1, \ldots, x_n) \mid\ & a \in A, \\ & x_1 \in X_1, \ldots, x_n \in X_n, \\ & \text{such that } S(x_1, \ldots, x_n) \\ & \text{and } p(a, x_1, \ldots, x_n)\}, \end{aligned}$$

where f is a function which is continuous on its first argument, $X_1, \ldots, X_n$ are arbitrary sets, $S(x_1, \ldots, x_n)$ is a formula which only depends on the $x_1, \ldots, x_n$, i.e., does not depend on a, and p is a predicate with the property that if $a < b$ and $p(a, x_1, \ldots, x_n)$ does not hold, then neither does $p(b, x_1, \ldots, x_n)$.

8 A Fixpoint Semantics

8.1 The Category of Processes

The category of processes, PROC, is $CP(\text{TRACE})$ where the objects are multisets of traces and arrows $G : a \to b$ are relations $G \subseteq a \times b$ such that for all $u \in b$ there is a unique $t \in a$ such that $\langle t, u\rangle \in G$, and when $\langle t, u\rangle \in G$, $t \leq u$. Since two traces are never related unless they have the same sequences of environments, arrows between processes will always link traces with the same environments.

The initial element of $CP(\text{TRACE})$ is the set of passive traces; $\{t \mid t \in \text{TRACE},\ r(t) = \emptyset\}$.

We have established that the operational semantics of our constraint language can be expressed compositionally in lemmas 11-13. The basic goal of this section is to translate these lemmas into a fixpoint semantics. Since we will use Lehmann's powerdomains, the semantic functions will have to be ω-functors. So for each statement α we define a semantic functor

$$\mathrm{M}[\![A]\!] : (\text{NAME} \to \text{PROC}) \to \text{PROC}$$

following the compositionality lemmas of the preceding chapter.

We will assume that each definition in the program is of the form $p(Y) :: A$, where $Y = \arg 1, \ldots, \arg |Y|$.

8.2 Parallel composition

An examination of lemma 11 gives us the following compositional formulation of the parallel composition.

$$\begin{aligned}\mathcal{O}_\Pi[\![\bigwedge_{i\in S} A_i]\!] = \{t \mid\ & t_i \in \mathcal{O}_\Pi[\![A_i]\!] \text{ and } v(t_i) = v(t) \text{ for } i \in S,\\ & r(t) = \textstyle\bigcup_{i\in S} r(t_i),\\ & \text{and } r(t_i) \cap r(t_j) = \emptyset, \text{ for } i, j \in S \text{ such that } i \neq j.\}\end{aligned}$$

We will perform the parallel composition in three steps. Given a family of ω-functors $F_i : C \to C_i$ for $i \in S$ we can create an ω-functor

$$\langle F_i\rangle_{i\in\omega} : C \to \prod_{i\in S} C_i,$$

so we have an ω-functor

$$\langle M[\![A_i]\!]\rangle_{i\in S} : (\text{NAME} \to \text{PROC}) \to \text{PROC}^S.$$

The functor CART : $CP(\text{TRACE})^S \to CP(\text{TRACE}^S)$ gives us the Cartesian product of the family of sets of traces from the subexpressions of the parallel composition. The function PARALLEL_S as defined below, takes a family of

sets of traces, where each set of traces is indexed by S, composes those that let themselves be composed, and discards the other.

$$\begin{aligned}\textsc{parallel}_S(T) = \{ \bigvee_{i \in S} v(t_i) \mid\ & \langle (v_i, r_i) \rangle_{i \in S} \in T, \\ & v_i = v_j, \text{ for } i, j \in S, \text{ and} \\ & r_i \cap r_j = \emptyset, \text{ for } i, j \in S \text{ such that } i \neq j\}.\end{aligned}$$

It is easy to see that the function $\textsc{parallel}_S$ can be defined as an ω-functor. So the semantics for the parallel composition can be given by the following equation.

$$\mathrm{M}[\![\bigwedge_{i \in S} A_i]\!] = \textsc{parallel}_S \circ \textsc{cart} \circ \langle M[\![A_i]\!] \rangle_{i \in S}$$

8.3 Closures

Lemma 14 describes the compositional behavior of the closure operator. For fixed X, Y, and c, this results in the ω-functor $\textsc{closure}(X, Y, c) : \textsc{proc} \to \textsc{proc}$ given by the following equation.

$$\begin{aligned}&\textsc{closure}(X, Y, c)(T) = \\ &\qquad \{f(u, (d_i)_{i \in \omega}, R) \mid\ u \in T, \\ &\qquad\qquad r(u) \subseteq R \subseteq \omega, \\ &\qquad\qquad (e_i)_{i \in \omega} = v(u), e_0 = c \sqcup I_X^Y(d_0), \\ &\qquad\qquad e_{i+1} = e_i \sqcup I_X^Y(d_{i+1}), \text{ for } i \in (\omega \setminus R), \text{and} \\ &\qquad\qquad d_{i+1} = I_Y^X(e_{i+1}) \sqcup d_i, \text{ for } i \in R\}\end{aligned}$$

where

$$\begin{aligned}f(u, (d_i)_{i \in \omega}, R) = \{i \in r(u) \mid\ & d_{i+1} = d_i, \text{ or} \\ & \text{there is no } j < i \text{ such that } j \in R \setminus r(u)\}.\end{aligned}$$

The semantics of the closure construct can now be defined as follows.

$$\mathrm{M}[\![A \langle c \rangle (X/Y)]\!] = \textsc{closure}(X, Y, c) \circ \mathrm{M}[\![\alpha]\!]$$

8.4 Selections

A selection has two behaviors; if a test succeeds, it will branch, if no test succeeds, it will do nothing. An ω-functor that captures the branching behavior for a constraint c should accept as argument a set of traces, and return those traces for which the constraint will be satisfied.

$$\begin{aligned}\textsc{branch}_c(T) = \{f(u, k) \mid\ & u \in T, k \in \omega, v(u)_0 \sqsupseteq c, \text{ and} \\ & (e_i)_{i \in \omega} \text{ a chain in } D_c \\ & \text{such that } e_{i+k+1} = v(u)_i \text{ for } i \in \omega\}\end{aligned}$$

where

$$f(u, k) = ((e_i)_{i \in \omega}, \{i + k + 1 \mid i \in r(u)\} \cup \{k\})$$

The second behavior of a selection operator occurs when no test ever succeeds. In this case, no reduction steps are ever performed. For fixed constraints $c_1, \ldots, c_n \in D_c$, we define the constant ω-functor $\textsc{sleep}_{c_1 \ldots c_n}$ over $CP(\textsc{trace})$ such that

$$\textsc{sleep}_{c_1 \ldots c_n}(T) = \{((d_i)_{i \in \omega}, \emptyset) \mid c_j \not\sqsubseteq d_k \text{ for any } j \leq n, k \geq 0\}$$

for objects T in $CP(\textsc{trace})$.

Now, the semantics of the selection operator can be given as follows.

$$\mathrm{M}[\![(c_1 \Rightarrow A_1 \,[]\, c_2 \Rightarrow A_2 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n)]\!] = (\bigoplus_{k \leq n} \textsc{branch}_{c_k} \circ \mathrm{M}[\![A_k]\!]) \oplus \textsc{sleep}_{c_1 \ldots c_n}$$

8.5 Procedures Calls

By the computation rules, a recursive call is reduced to a closure containing the body of its definition.

$$\mathrm{M}[\![p(X)]\!]\sigma = \textsc{closure}(X, Y, c)(\sigma(p)),$$

where $Y = \arg 1, \ldots, \arg |X|$. Remember that each procedure definition is required to be of the form $p(\arg 1, \ldots, \arg n) :: A$.

8.6 Tell Constraints

Whenever the tell constraint c has performed a computation step, the environment must be at least as strong as c.

$$\mathrm{M}[\![c]\!]\sigma = \{t \in \textsc{trace} \mid v(t)_{i+1} = v(t)_i \sqcup c, \text{ for } i \in r(t), \text{ and } c \sqsubseteq \sqcup v(t)\}.$$

9 Abstraction

In this chapter, a simple abstraction operator is defined and proved to give full abstraction.

A *consequence operator*[1] is a function $f : D \to D$ such that for all $x \in D$, $f(x) \sqsupseteq x$ and $f(f(x)) = f(x)$. The *functionality* of a trace t is defined to be the least continuous consequence operator $\mathrm{fn}(t) : D \to D$ such that $i \in r(t)$ implies $\mathrm{fn}(t)(x_i) \sqsupseteq x_{i+1}$, where $(x_i)_{i \in \omega} = v(t)$. A trace t is a *subtrace of* a trace u if $\mathrm{fn}(t) \sqsubseteq \mathrm{fn}(u)$ and $\sqcup v(t) = \sqcup v(u)$.

Note that the subtrace relation is not a partial order, that is, two traces may be subtraces of each other without being equal, however, if $t \leq u$ then t is a subtrace of u.

[1] Saraswat *et al.* used this type of functions for a similar purpose and called them *closure operators* [10]. We chose the term consequence operator to avoid any confusion with the syntactic construct.

For a set of traces p, let

$$\lfloor p \rfloor = \{t' \mid t' \text{ is a subtrace of } t \in p\}.$$

The idea is that $\lfloor \cdot \rfloor$ should serve as an abstraction operation. This is expressed in the following theorem.

Theorem 17 Suppose we have a program Π and two statements A and B such that $\lfloor \mathcal{O}_\Pi[\![A]\!] \rfloor \neq \lfloor \mathcal{O}_\Pi[\![B]\!] \rfloor$. There is an agent C, and an $x \in D_c$ such that $\mathcal{R}_\Pi[\![A \wedge C]\!]x \neq \mathcal{R}_\Pi[\![B \wedge C]\!]x$.

To prove the theorem, we need to give a definition and establish a few properties of traces.

For a continuous function $f : D \to D$, define $\mathrm{fix}(f) : D \to D$ such that for $x \in D$, $\mathrm{fix}(f)x$ is the smallest $y \sqsupseteq x$ such that $f(y) = y$. Note that for functions $f, g : D \to D$ such that $f, g \sqsupseteq \mathbf{id}$, $\mathrm{fix}(f \circ g) \sqsupseteq f, g$.

Lemma 18 For two traces t and u such that $t \vee u$ is defined, $\mathrm{fn}(t \vee u) = \mathrm{fix}(\mathrm{fn}(t) \circ \mathrm{fn}(u))$.

For a pair of constraints c, d the operational semantics of the agent $(c \Rightarrow d)$ consists of traces t such that whenever $i \in r(t)$, and $v(t)_i \sqsupseteq c$, then $v(t)_{i+1} \sqsubseteq d \sqcup v(t)_i$, and there is a $j \geq i$ such that $v(t)_j \sqsupseteq d$. For a function f such that

$$f(x) = \begin{cases} x \sqcup d, & \text{if } c \sqsubseteq x; \\ \bot, & \text{otherwise,} \end{cases}$$

we have $\mathrm{fn}(t) \sqsubseteq f$.

So given a set of pairs $(c_i, d_i)_{i \in S}$, we can construct a (potentially infinite) conjunction

$$\bigwedge_{i \in S} (c_i \Rightarrow d_i),$$

such that each trace t in the operational semantics of the conjunction has a functionality which is weaker or equal to than the function g, given by the definition

$$g(x) = \mathrm{fix}(\sqcup\{d_i \mid c_i \sqsubseteq x,\ i \in S\}).$$

Proof. Assume a program Π and suppose that A and A' are two statements such that $\lfloor \mathcal{O}_\Pi[\![A]\!] \rfloor = p$ and $\lfloor \mathcal{O}_\Pi[\![A']\!] \rfloor = p'$, but there is a trace t such that $t \in p$ and $t \notin p'$. Let u be the trace $(v(t), \omega - r(t))$. Let C be the agent

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

From the reasoning above it should be clear that any trace of C which has the same limit as u is a subtrace of u. Consider $A \wedge C$ and $A' \wedge C$. Since $u \in \mathcal{O}_\Pi[\![A]\!]$ and $t \vee u$ is a maximal trace we have $y \in \mathcal{R}[\![A \wedge C]\!]x$, where $x = v(t)_0$ and $y = \sqcup v(t)$. Suppose that $y \in \mathcal{R}_\Pi[\![A' \wedge C]\!]x$. This means that

there has to be a trace $t' \in p'$ such that $\sqcup v(t_0) = x$ and $u' \in \mathcal{O}_\Pi[\![C]\!]$ such that $t' \vee u'$ is maximal.

By lemma 18, $\mathrm{fn}(t \vee u) = \mathrm{fix}(\mathrm{fn}(t) \circ \mathrm{fn}(u))$, and the way u was selected guarantees that $\mathrm{fn}(t)$ is minimal in the sense that if $\mathrm{fix}(f \circ \mathrm{fn}(u)) = \mathrm{fix}(\mathrm{fn}(t) \circ \mathrm{fn}(u))$ for any consequence operator f, then $\mathrm{fn}(t) \sqsubseteq f$. Also note that $\mathrm{fn}(t \vee u)$ is completely defined by the 'endpoints' x and y, so that

$$\mathrm{fn}(t \vee u)z = \begin{cases} z \sqcup y, & \text{if } z \sqsupseteq x \\ z, & \text{otherwise.} \end{cases}$$

Since $t' \vee u'$ has the same endpoints, we have $\mathrm{fn}(t' \vee u') = \mathrm{fn}(t \vee u)$. Since u' is a subtrace of u, $\mathrm{fn}(u') \sqsubseteq \mathrm{fn}(u) \sqsubseteq \mathrm{fn}(t \vee u) = \mathrm{fix}(\mathrm{fn}(t') \circ \mathrm{fn}(u'))$, so $\mathrm{fix}(\mathrm{fn}(t') \circ \mathrm{fn}(u')) = \mathrm{fix}(\mathrm{fn}(t') \circ \mathrm{fn}(u))$. So $\mathrm{fix}(\mathrm{fn}(t) \circ \mathrm{fn}(u)) = \mathrm{fix}(\mathrm{fn}(t') \circ \mathrm{fn}(u))$. But the minimality of $\mathrm{fn}(t)$ implies that $\mathrm{fn}(t) \sqsubseteq \mathrm{fn}(t')$ which contradicts our original assumption. □

10 Summary

In this paper, we presented a compositional operational semantics for a class of non-deterministic concurrent constraint languages, where the semantics of an agent was given as a set of traces. Since concurrent constraint languages allow recursive definitions, its is necessary to find a suitable form of powerdomains where fixpoints of continuous functions exist. There is a wide range of powerdomain constructions, but the only suitable for our purpose seems to be Lehmann's powerdomains. Other powerdomains cannot represent all sets, or do not allow the function $f(x) = \{x\}$ as a continuous function.

It turns out that many semantic functions over a powerdomain fit into a general pattern, and it is quite easy to establish that functions belonging to this pattern are continuous in Lehmann's powerdomain theory. However, since Lehmann's construction is based on category theory, the fixpoint obtained is the *initial* fixpoint. Even though it intuitively appears to be clear that it is the desired fixpoint, it is not obvious how to relate this fixpoint to the inductive definitions of the computation rules and the fairness property.

We also gave the definition of a simple abstraction operator which gives a fully abstract semantics, since two agents which are mapped to different values by the abstraction operator can be put into a context where the set of possible computations differ. It remains to be proved that the abstraction operator preserves compositionality.

References

[1] S. Abramsky, On semantic foundations for applicative multiprogramming, *Proceedings of the Tenth ICALP*, Springer LCNS 154, 1-14, New York 1983.

[2] K. R. Apt, G.D. Plotkin, Countable nondeterminism and random assignment, *JACM*, 33(4):724-767, 1986.

[3] F.S. de Boer, C. Palamidessi, A fully abstract model for concurrent constraint programming, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Springer LNCS 493, 296-319, 1991.

[4] G. Kahn, The semantics of a simple language for parallel programming, *Proceedings of IFIP Congress 74*, North-Holland, 471-475, 1974.

[5] D. J. Lehmann, Categories for fixed-point semantics, *17th Annual Symposium on Foundations of Computer Science*, 122-126, 1976.

[6] D. J. Lehmann, *Categories for Fixed-point Semantics*, PhD thesis, Hebrew University of Jerusalem, 1976.

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

[8] P. Panangaden, J. R. Russell, A category-theoretic semantics for unbounded nondeterminacy, *Mathematical Foundations of Programming Semantics, Proceedings, 1989*, Springer LNCS 442, 1990.

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

[10] V. A. Saraswat, M. Rinard, P. Panangaden, Semantic foundations of concurrent constraint programming, *Proceedings of the 8'th ACM Symposium on Principles of Programming languages*, 333-352, 1991.

[11] D. S. Scott, Domains for denotational semantics, *ICALP'82*, Springer LNCS 140, 577-613, 1982.

[12] M. B. Smyth, Powerdomains, *Journal of Computer and System Sciences*, 16:23-36, 1978.