Analele Ştiinţifice ale Universităţii "Alexandru Ioan Cuza" din Iaşi
Informatică, Tomul XV, 2004, 22–37

Scientific Annals of the "Alexandru Ioan Cuza" University of Iaşi
Computer Science Section, Tome XV, 2004, 22–37

Continuation Semantics for Concurrency and Concurrent Languages Prototyping

Eneia Todoran[1]
Nikolaos Papaspyrou[2]
Florian Boian[3]
Cosmina Ivan[1]

Abstract

This paper shows that, by using the "continuation semantics for concurrency" (CSC) technique (recently introduced by us), denotational semantics can be used not only as a method for formal specification and design, but also as a general method for compositional prototyping of concurrent programming languages. In this new approach, a denotational function uses continuations to produce incrementally a stream of observables, i.e. a single execution trace, rather than an element of some powerdomain construction. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. In this paper we employ classic (cpo-based) domains and we develop denotational models for two concurrent languages of progressive complexity. The first language provides a construct for parallel composition; the second language extends the first one with primitives for CSP-like synchronous communication. The both models are designed by using the CSC technique.

[1]Technical University of Cluj-Napoca, Department of Computer Science, Baritiu Str. 28, (400027) Cluj-Napoca, ROMANIA; Phone/Fax: +40-264-401221 / +40-264-594491; E-mail: {Eneia.Todoran , Cosmina.Ivan}@cs.utcluj.ro

[2]National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, Polytechnioupoli, 15780 Zografou, Athens, GREECE; Phone/Fax: +30-1-7722486 / +30-1-7722519; E-mail: nickie@softlab.ntua.gr

[3]"Babes-Bolyai" University of Cluj-Napoca, Faculty of Mathematics and Computer Science, Department of Computer Science, M. Kogalniceanu Str. 1, Cluj-Napoca, ROMANIA; Phone/Fax: +40-264-405300 / +40-264-591906; E-mail: florin@cs.ubbcluj.ro

1 Introduction

Classic continuations and variations have proven useful for modeling a variety of control structures, such as non-local exits, coroutines, multitasking [17], tree-structured concurrency [5], synchronous and asynchronous communication [13]. We have recently introduced a variant of the continuations technique, named *continuation semantics for concurrency* (CSC) [13, 16, 14], which provides excellent flexibility in the denotational modeling of concurrent systems [16, 14]. The CSC technique provides a "pure" continuation-based approach to communication and concurrency in which all control structures - including the ones encountered in communicating systems - are modeled as operations manipulating continuations. Intuitively, it is a semantic formalization of a process scheduler simulated on a sequential machine. A continuation in the CSC approach is a *configuration* (e.g. a multiset) of computations (partially evaluated denotations). Every moment there is only one *active* computation; subsequently, another computation taken from the continuation is planned for execution. In this way it is possible to obtain the desired interleaving behavior for parallel composition.

A prototype, is an initial version of a system which serves as a mechanism for identifying and specifying system requirements. It is easy to use a functional language and classic denotational semantic techniques to produce prototype implementations for (various aspects of) sequential programming languages (cf. also the early work of Peter Mosses on the use of denotational descriptions in compiler generation [9, 10]). However, classic techniques seem to fail in producing compositional prototypes for concurrent languages. In this paper we show that by employing the CSC technique, denotational semantics can be used not only as a method for formal specification and design, but also as a general method for compositional prototyping of concurrent programming languages. In this new approach (introduced by us in [15]) a denotational function uses continuations to produce incrementally a stream of observables, i.e. a single execution trace, rather than an element of some powerdomain construction. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. The immediate implementation of such a denotational model in an appropriate functional language is a compositional prototype for the concurrent programming language under study.

In [15], we showed how to use the CSC technique in designing a denotational (compositional) interpreter for a simple CSP-like language [6, 7] providing constructs for parallel composition, synchronous communication, and a primitive for producing intermediate outputs (observables) at the standard output file. The interpreter was implemented in the functional programming language Haskell [4]. The compositional interpreter given in [15] generates incrementally a single stream of observables and it simulates the selection of an arbitrary execution trace by using a (pseudo-)random number generator.

The CSC technique was introduced in [13, 14] using metric semantics [2]. In [16, 15] the denotational models were developed using Haskell. In this paper we employ classic (cpo-based) domains and we develop denotational semantic models for two simple concurrent languages of progressive complexity. The first language provides a construct for parallel composition; the second language extends the first one with primitives for CSP-like synchronous communication. The second denotational model is a mathematical formalization of the compositional interpreter given in [15]. As it will be seen, when the CSC technique is employed in this mathematical framework no communication attempts or silent steps need to be produced as final yields of a denotational semantics. Throughout this paper, we rely on the mathematical apparatus and notation in [12]. To the best of our knowledge, denotational semantics have never been used systematically as a method for concurrent languages prototyping and all our attempts to solve the problem by using only classic compositional techniques have failed.

2 A simple language with parallel composition

In this section we consider a very simple imperative concurrent language, called L. The semantics of L is given here without mathematical justifications. The mathematical justifications are postponed to sections 3 and 4. In section 3 we give a denotational semantic model for a language L_{CSP} which strictly extends L with CSP-like synchronous communication, and in section 4 we solve our main domain equation.

We assume given a set $(v \in)Var$[4] of (numerical) variables, a set $(e \in)Exp$ of numerical expressions, a set $(b \in)BExp$ of boolean expressions, and a set $(x \in)PVar$ of procedure variables. The syntax of L is given in below in BNF.

[4]The notation $(x, y, ... \in)X$ introduces the set X with typical variables $x, y, ...$. Whenever we use a set in a context where a domain is needed, we assume it is equipped with the discrete order.

Definition 1 *(Syntax of L)*
The set $(s \in)Stmt$ of statements in L is given by the following grammar:

$$s(\in Stmt) ::= \textbf{skip} \mid a; s \mid \textbf{if } b \textbf{ then } s \textbf{ else } s \mid s \parallel s \mid \\ \textbf{call}(x) \mid \textbf{letrec } x \textbf{ be } s \textbf{ in } s,$$

where the set of elementary actions is defined by:

$$a ::= v := e \mid \textbf{write}(e)\,.$$

L provides assignment ($v := e$), a primitive for writing the value of a numerical expression at the standard output file (**write**(e)), a null command (**skip**), sequential composition (in the form of action prefixing: $a; s$), a conditional command (**if** b **then** s **else** s), parallel composition ($s \parallel s$), and recursion.

The meaning of expressions is defined by two valuations $\mathcal{E}[\![\cdot]\!] : Exp \to (State \to \mathbb{N})$ and $\mathcal{B}[\![\cdot]\!] : BExp \to (State \to Bool)$, where $(\sigma \in)State = Var \to \mathbb{N}$ is the domain of *states*.

According to our design decision, the denotational semantics should produce arbitrary execution traces of concurrent programs. The final yield of the denotational semantics is a sequence of "observables" (in our case natural numbers $\in \mathbb{N}$), which is an element of the following (recursively defined) domain: $O \cong (\{\epsilon\} + (\mathbb{N} \times O))_\perp$, where ϵ is the empty sequence. O is a lifted domain; the bottom element ($\perp$) represents a non-terminating computation that produces no observable effect.

To simulate the nondeterministic behavior of a concurrent system, the denotational mapping uses a (pseudo-)random number generator; random numbers are natural numbers. We put $(\rho \in)R = \mathbb{N}$ and we assume given an initial random number $\rho_0(\in R)$ and a mapping $r : R \to R$ that produces a new random number from a given one[5].

In the definition of the denotational semantics for L we employ the following domain: $(\phi \in)D = Cont \to R \to State \to O$, where $Cont$ is the domain of *continuations* which, in the CSC approach, is a configuration (e.g. a multiset) of computations. Here, we implement this concept as follows: $(\gamma \in)Cont = Id \times Kont$, $(\vartheta \in)Kont = Id \to Proc$, and $(p \in)Proc = Sched \times D$. The elements of Id are *process identifiers*. In the present setting it is convenient to put: $(\iota \in)Id = \mathbb{N}$. An element $\vartheta \in Kont$ is a multiset of processes, where each process is a pair containing

[5]A very simple example of such a pseudo-random number generator can be defined as follows: $\rho_0 = 17489$ and $r(\rho) = (25173 \times \rho + 13849)\,\%\,65536$.

some scheduling information and a computation (an element of type D). A continuation $\gamma = (\bar{\iota}, \vartheta) \in Cont$ implements a dynamic pool of processes; only elements $\vartheta(\iota)$ for $\iota < \bar{\iota}$ are handled by the semantic functions, and $\bar{\iota}$ always points to the next *free location* in ϑ. This means that continuations are *finite* structures.

In this section the domain *Sched* is defined by: $(\varsigma \in) Sched = \{null\} + \{proc\}$. Finally, to deal with recursion we define *semantic environments* as follows: $(\eta \in) Env = PVar \to D$.

We see that, in order to get a good mathematical foundation for our design, we need to solve the following domain equation (where Id, $Sched$ and E do not depend on D): $D \cong (Id \times (Id \to (Sched \times D))) \to E$, where $E = R \to State \to O$ is a domain with least element $(\bot_E = \lambda\rho.\lambda\sigma.\bot_O)$, which means that D also has a least element; more precisely $\bot_D = \lambda\gamma.\bot_E$. In section 3 we will employ a semantic domain defined by a very similar equation; in fact, only the domains *Sched* and O are slightly modified in section 3, but the both equations can be solved in the same way. The solution is given in section 4.

In the definition of the denotational semantics we use a predicate $terminates : Cont \to Bool$, given by

$$terminates(\bar{\iota}, \vartheta) = \ \mathsf{if}\ (\bar{\iota} = 0)\ \mathsf{then}\ true\ \mathsf{else}\ \bigwedge_{0 \le \iota < \bar{\iota}} isnull(\vartheta(\iota));$$

this predicate formalizes the intuitive notion of *termination* for a continuation. Here, $isnull : Proc \to Bool$ is given by: $isnull(\varsigma, \phi) = (\varsigma = null)$.

Only processes $(\varsigma, \phi) \in Proc$ such that $\varsigma \neq null$ are planned for execution. In the definition of the denotational semantics, we assume given a mapping $sched : (Cont \times R) \to Id$ which uses a random number to simulates an arbitrary selection of a process in a continuation γ such that $\neg terminates(\gamma)$. For a given (non-terminating) continuation $\gamma = (\bar{\iota}, \vartheta)$ and random number ρ, $sched(\gamma, \rho)$ will return a (randomly chosen) process identifier $\iota \in Id$ such that $\iota < \bar{\iota}$, and $\vartheta(\iota) = (proc, \phi)$ for some $\phi \in D$. A detailed design for such a scheduling mapping is given in section 3 for a more complex language with synchronous communication.

Definition 2 *(Denotational prototype semantics for L)*

(a) We define $\kappa (\in D)$ as follows:

$$\kappa(\bar{\iota}, \vartheta)(\rho)(\sigma) =$$

$$\begin{cases} \epsilon, & \text{if } terminates(\bar{\iota},\vartheta) \\ \phi\,(\bar{\iota},(\vartheta \mid \iota \mapsto (null,\phi))(r\rho)(\sigma), & \text{if } \neg terminates(\bar{\iota},\vartheta) \text{ and} \\ & \quad sched((\bar{\iota},\vartheta),\rho) = \iota \text{ and} \\ & \quad \vartheta(\iota) = (proc,\phi) \end{cases}$$

(b) *The denotational semantics* $[\![\cdot]\!] : Stmt \to Env \to D$ *is defined as follows:*

$$[\![\,\mathbf{skip}\,]\!]\eta\gamma\rho\sigma = \kappa\gamma\rho\sigma$$

$$[\![\,v := e; s\,]\!]\eta\gamma\rho\sigma = \kappa((proc,\,[\![\,s\,]\!]\eta)\ ::\ \gamma)\rho\,(\sigma \mid v \mapsto \mathcal{E}[\![\,e\,]\!]\sigma)$$

$$[\![\,\mathbf{write}(e)\,;s\,]\!]\eta\gamma\rho\sigma = (\,\mathcal{E}[\![\,e\,]\!]\sigma, \kappa((proc,\,[\![\,s\,]\!]\eta)\ ::\ \gamma)\rho\sigma)$$

$$[\![\,\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\,]\!]\eta\gamma\rho\sigma =$$
$$\qquad \text{if } \mathcal{B}[\![\,b\,]\!]\sigma \text{ then } [\![\,s_1\,]\!]\eta\gamma\rho\sigma \text{ else } [\![\,s_2\,]\!]\eta\gamma\rho\sigma$$

$$[\![\,\mathbf{call}(x)\,]\!]\eta\gamma\rho\sigma = \eta(x)\gamma\rho\sigma$$

$$[\![\,\mathbf{letrec}\ x\ \mathbf{be}\ s_1\ \mathbf{in}\ s_2\,]\!]\eta\gamma\rho\sigma =$$
$$\qquad [\![\,s_2\,]\!]\,(\eta \mid x \mapsto fix(\lambda\phi.\,[\![\,s_1\,]\!]\,(\eta \mid x \mapsto \phi)))\gamma\rho\sigma$$

$$[\![\,s_1 \parallel s_2\,]\!]\eta\gamma\rho\sigma =$$
$$\qquad \begin{cases} [\![\,s_1\,]\!]\eta((proc,\,[\![\,s_2\,]\!]\eta)\ ::\ \gamma)(r\rho)\sigma, & \text{if } \rho\%2 = 0 \\ [\![\,s_2\,]\!]\eta((proc,\,[\![\,s_1\,]\!]\eta)\ ::\ \gamma)(r\rho)\sigma, & \text{if } \rho\%2 = 1 \end{cases}$$

In this (and subsequent) definitions % *is the* modulo *operator; Also, we have used the notation* $p\ ::\ (\bar{\iota},\vartheta) \overset{not.}{=} (newId(\bar{\iota}),(\vartheta \mid \bar{\iota} \mapsto p))$, *with* $newId(\iota) = \iota + 1$, *for any* $p \in Proc$ *and for any* $\gamma = (\bar{\iota},\vartheta) \in Cont$.

The first three clauses in the definition of $[\![\cdot]\!]$ define the behavior of primitive commands. Only a **write**(e) statement produces an observable result. Assignments and null (i.e. **skip**) commands do not contribute to the final yield of $[\![\cdot]\!]$. However, unlike **skip** and **write**(e), an assignment command changes the current state. The second and third clauses in the definition of $[\![\cdot]\!]$ treat action prefixing constructs of the form $a; s$ (where a can be either an assignment or a **write**(e) statement); in both cases the denotation of s is added to the continuation. The following three clauses define the semantics of conditional statements and recursion in a standard manner. Following the CSC approach [13, 16], the semantics of parallel composition is based on a nondeterministic choice (implemented here by using the random number generator) between two alternative computations: one starting from the first statement and the other starting from the second one. The denotational semantics does not only depend on "traditional arguments" - such as semantic environment, continuation, and state - but also on a random number, that is used to simulate the nonterministic behaviour of a "real" concurrent system, by choosing at random a single execution trace.

In the definition above fix is the classical fixed point operator[6]. It is not difficult to see that, provided *terminates* and *sched* are continuous mappings, κ and $[\![\cdot]\!]$ are also continuous. We postpone well-definedness and continuity issues to section 3 where we define the semantics of a language that strictly subsumes L.

3 A CSP-like language

In this section we consider a language, called L_{CSP}, that extends L with CSP-like synchronous communication. Let $(v\in)Var$, $(e\in)Exp$, $(b\in)BExp$, and $(x\in)PVar$ be as in section 2. In defining the syntax of L_{CSP} we also employ a set $(c\in)Chan$ of *communication channels.*

Definition 3 *(Syntax of L_{CSP})*
The set of statements in L_{CSP} is given by the following grammar:

$$\begin{aligned} s(\in Stmt) ::= \; & \mathbf{skip} \mid a;s \mid \mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s \mid s \parallel s \mid \\ & \mathbf{call}(x) \mid \mathbf{letrec}\ x\ \mathbf{be}\ s\ \mathbf{in}\ s, \end{aligned}$$

where: $\quad a ::= v := e \mid \mathbf{write}(e) \mid c!e \mid c?v.$

All constructs, apart from $c!e$ and $c?v$ are as in section 2. The constructs $c!e$ and $c?v$ are as in Occam [8]. Synchronized execution of two actions $c!e$ and $c?v$, occurring in parallel processes, results in the transmission of the value of the expression e along the channel c from the process executing the $c!e$ statement to the process executing the $c?v$ statement. The latter assigns the received value to the variable v.

As in section 2, we use a domain $(\sigma\in)State = Var \to \mathbb{N}$ of states, and valuations $\mathcal{E}[\![\cdot]\!] : Exp \to (State \to \mathbb{N})$, and $\mathcal{B}[\![\cdot]\!] : BExp \to (State \to Bool)$, for numerical and boolean expressions. In the present setting, it is also convenient to let ξ range over $(\xi\in)\ State \to \mathbb{N}$.

The domain that we employ in the definition of the denotational (prototype) semantics for L_{CSP} is very similar to the one that we have used for

[6]If $f : X \to X$ is a continuous mapping and X is a domain with least element $\perp_X$ then $fix : (X \to X) \to X$ is defined as follows: $fix(f) = \bigsqcup_{i\in\omega} f^i(\perp_X)$. It is well-known that fix is a continuous mapping, and that $fix(f)$ is the least fixed point of f.

L. In fact, only the definitions for domains O and $Sched$ are more elaborate than they were in section 2. In the present setting, O reflects the possibility of an 'abnormal' termination represented by a new constant δ interpreted as *deadlock*, and the information embodied in $Sched$ allows us to represent (pairs of) communicating processes. Apart from these changes, all domain definitions look as in section 2. However, to avoid ambiguities we give here all domain definitions for L_{CSP}:

$$(\phi \in)D = Cont \to R \to State \to O,$$

$$(\gamma \in)Cont = Id \times Kont, \qquad (\vartheta \in)Kont = Id \to Proc,$$

$$(p \in)Proc = Sched \times D, \qquad O \cong (\{\epsilon\} + \{\delta\} + (\mathbb{N} \times O))_{\perp},$$

$$(\varsigma \in)Sched = \{null\} + \{proc\} + Snd + Rcv,$$

$$Snd = Chan \times (State \to \mathbb{N}), \qquad Rcv = Chan \times Var, \text{ and}$$

$$(\eta \in)Env = PVar \to D.$$

O is a domain with least element, which easily implies that both $R \to State \to O$ and D are domains with least elements. It is not difficult to see that, the above definitions lead us to the domain equation that is solved in section 4. We emphasize that, apart from $O, D, Proc, Cont, Kont$ and Env, all domains that are employed in the semantic constructions given in this section are discretely ordered.

The interpretation of the above definitions is similar to the one given in section 2 for L. Thus, D is the domain of *computations*, $Cont$ is the domain of *continuations*, O is used as final yield for the denotational mapping, $Sched$ embodies scheduling information, and Env is the domain of *semantic environments* that helps us deal with recursion. Continuations are again "finite structures". More precisely, for any continuation $\gamma = (\bar{\iota}, \vartheta) \in Cont$, only elements $\vartheta(\iota)$ for $\iota < \bar{\iota}$ are handled by the semantic functions, and $\bar{\iota}$ always points to the next *free location* in ϑ. Moreover, we put again $(\iota \in)Id = \mathbb{N}$, $(\rho \in)R = \mathbb{N}$, and we use the random number generator (a mapping $r : R \to R$ together with some 'initial' random number ρ_0) as given in section 2. However, the new definitions for O and $Sched$ show that, computations specified in L_{CSP} may end in deadlock, and that continuations may contain communication attempts. The scheduling of processes needs to be more elaborate in the present setting.

In the present setting we introduce some new notations. For easier readability, we denote typical elements (c, ξ) of Snd by $c!\xi$, and we denote typical elements (c, v) of Rcv by $c?v$. Also, for any $\gamma = (\bar{\iota}, \vartheta) \in Cont$ we will use the following abbreviations: $\gamma[\iota] \overset{not.}{=} \vartheta(\iota)$, $\gamma\langle\iota\rangle \overset{not.}{=}$ $\mathsf{let}\ (\varsigma, \phi) = \vartheta(\iota)\ \mathsf{in}\ \phi$, and $\langle\gamma \mid \iota_1 \mapsto p_1 \mid ... \mid \iota_n \mapsto p_n\rangle \overset{not.}{=} (\bar{\iota}, (\vartheta \mid \iota_1 \mapsto p_1 \mid ... \mid \iota_n \mapsto p_n))$.

As shown in [15], in the CSC approach the information contained in continuations suffices for all process scheduling purposes, including process synchronization, termination and deadlock detection. We first formalize the intuitive notion of *termination* for L_{CSP} by defining (as in the previous section) a predicate $terminates : Cont \to Bool$, $terminates(\bar{\iota}, \vartheta) =$ $\mathsf{if}\ (\bar{\iota} = 0)\ \mathsf{then}\ true\ \mathsf{else}\ \bigwedge_{0 \leq \iota < \bar{\iota}} isnull(\vartheta(\iota))$, where $isnull : Proc \to Bool$ is given by $isnull(\varsigma, \phi) = (\varsigma = null)$. It is easy to prove the continuity of *terminates*, but we defer the issue to 3.1.

For scheduling purposes, it is also convenient to introduce the following auxiliary domain:

$$(\pi \in)\Pi = \{nil\} + Id + Id \times Id \times (State \to \mathbb{N}) \times Var.$$

We assume given a continuous mapping $sched : (Cont \times R) \to \Pi$ that uses a random number $\rho (\in R)$ to model a random choice of a process or of a pair of communicating processes in a continuation $\gamma (\in Cont)$. More precisely, the mapping $sched(\gamma, \rho)$ behaves as follows. It either (1) chooses at random a process identifier $\iota \in Id$ such that $\gamma[\iota] = (proc, \phi)$ for some $\phi \in D$, or (2) it chooses at random a pair of process identifiers $\iota_1, \iota_2 \in Id$ such that $\gamma[\iota_1] = (c!\xi, \phi_1)$ and $\gamma[\iota_2] = (c?v, \phi_2)$, for some $c \in Chan$, $\xi (\in State \to \mathbb{N})$ and $v (\in Var)$, in which case the components v and ξ (of the distributed assignment that is performed upon synchronization) are returned together with the process identifiers ι_1 and ι_2, or, (3) when none of the above choices are possible, it returns nil, which signifies *deadlock* detection. Section 3.1 offers an example of such a function *sched*.

Apart from the clauses for synchronous communication, the denotational function $[\![\cdot]\!]$ for L_{CSP} looks very similar to the one given in section 2 for L; however, the semantic domains are different and the auxiliary mapping κ is more complex.

Definition 4 *(Denotational prototype semantics for L_{CSP})*

(a) We define $\kappa (\in D)$ as follows:

$$\kappa = fix(K),$$

with $K : D \to D$ given by:

$$K(k)(\gamma)(\rho)(\sigma) =$$
$$\begin{cases} \epsilon, & \text{if } terminates(\gamma) \\ \delta, & \text{if } \neg terminates(\gamma) \text{ and } sched(\gamma,\rho) = nil \\ \gamma\langle\iota\rangle \, \langle \gamma \mid \iota \mapsto \overline{\gamma[\iota]} \rangle \; (r\rho)(\sigma), & \\ \quad \text{if } \neg terminates(\gamma) \text{ and } sched(\gamma,\rho) = \iota & \\ k \; \langle \gamma \mid \iota_1 \mapsto \widetilde{\gamma[\iota_1]} \mid \iota_2 \mapsto \widetilde{\gamma[\iota_2]} \,\rangle \, (r\rho)(\sigma \mid v \mapsto \xi\, \sigma), & \\ \quad \text{if } \neg terminates(\gamma) \text{ and } sched(\gamma,\rho) = (\iota_1, \iota_2, \xi, v) & \end{cases}$$

where we have used the following notations: $\widetilde{(\varsigma,\phi)} \overset{not.}{=} (proc, \phi)$, *and* $\overline{(\varsigma,\phi)} \overset{not.}{=} (null, \phi)$.

(b) *The denotational semantics* $[\![\cdot]\!] : Stmt \to Env \to D$ *is defined as follows:*

$$[\![\, \mathbf{skip} \,]\!]\eta\gamma\rho\sigma = \kappa\gamma\rho\sigma$$
$$[\![\, v := e; s \,]\!]\eta\gamma\rho\sigma = \kappa((proc, [\![\, s \,]\!]\eta) \;::\; \gamma)\rho\,(\sigma \mid v \mapsto \mathcal{E}[\![\, e \,]\!]\sigma)$$
$$[\![\, \mathbf{write}(e)\,; s \,]\!]\eta\gamma\rho\sigma = (\mathcal{E}[\![\, e \,]\!]\sigma, \kappa((proc, [\![\, s \,]\!]\eta) \;::\; \gamma)\rho\sigma)$$
$$[\![\, c!e; s \,]\!]\eta\gamma\rho\sigma = \kappa((c!\,\mathcal{E}[\![\, e \,]\!]\,,\, [\![\, s \,]\!]\eta) \;::\; \gamma)\rho\sigma$$
$$[\![\, c?v; s \,]\!]\eta\gamma\rho\sigma = \kappa((c?v,\, [\![\, s \,]\!]\eta) \;::\; \gamma)\rho\sigma$$
$$[\![\, \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \,]\!]\eta\gamma\rho\sigma =$$
$$\text{if } \mathcal{B}[\![\, b \,]\!]\sigma \text{ then } [\![\, s_1 \,]\!]\eta\gamma\rho\sigma \text{ else } [\![\, s_2 \,]\!]\eta\gamma\rho\sigma$$
$$[\![\, s_1 \parallel s_2 \,]\!]\eta\gamma\rho\sigma =$$
$$\begin{cases} [\![\, s_1 \,]\!]\eta((proc, [\![\, s_2 \,]\!]\eta) \;::\; \gamma)(r\rho)\sigma, & \text{if } \rho\%2 = 0 \\ [\![\, s_2 \,]\!]\eta((proc, [\![\, s_1 \,]\!]\eta) \;::\; \gamma)(r\rho)\sigma, & \text{if } \rho\%2 = 1 \end{cases}$$
$$[\![\, \mathbf{call}(x) \,]\!]\eta\gamma\rho\sigma = \eta(x)\gamma\rho\sigma$$
$$[\![\, \mathbf{letrec}\ x\ \mathbf{be}\ s_1\ \mathbf{in}\ s_2 \,]\!]\eta\gamma\rho\sigma =$$
$$[\![\, s_2 \,]\!]\,(\eta \mid x \mapsto fix(\lambda\phi.\, [\![\, s_1 \,]\!]\,(\eta \mid x \mapsto \phi)))\gamma\rho\sigma$$

where we have used the notation:

$$p \;::\; (\bar{\iota}, \vartheta) \overset{not.}{=} (newId(\bar{\iota}), (\vartheta \mid \bar{\iota} \mapsto p)),$$

with $newId(\iota) = \iota + 1$, *for any* $p \in Proc$ *and* $\gamma = (\bar{\iota}, \vartheta) \in Cont$.

(c) *Moreover, we can define a mapping* $\mathcal{D}[\![\cdot]\!] : Stmt \to State \to O$ *that computes a possible execution trace for any statement evaluated in any state as follows:*

$$\mathcal{D}[\![s]\!] = [\![s]\!] \eta_0 (0, \vartheta_0) \rho_0,$$

where η_0 and $\gamma_0 = (0, \vartheta_0)$ are "initial" values for the semantic environment and continuation, and ρ_0 is the initial random number; remark that $terminates(\gamma_0) = true$, but there is no need to impose constraints on η_0 or ϑ_0.

It is easy to see that (provided *terminates* and *sched* are continuous mappings) K is a continuous mapping and thus κ is well-defined and continuous, and $[\![\cdot]\!]$ is also continuous.

3.1 A process scheduler with random choice

The specification that we gave in section 3 does not determine a unique function $sched : (Cont \times R) \to \Pi$. In this section we only present a possible design for *sched*.

All operations involved in process scheduling are essentially iterations on continuations. The basic idea is that, given a continuation $\gamma = (\bar{\iota}, \vartheta)$, we need to process somehow the information contained in all elements $\vartheta(\iota)$, for $0 \leq \iota < \bar{\iota}$. In fact, the type of ϑ is $Id \to Proc$, but we only need to be able to process various derived information embodied in functions of types $Id \to Bool$, $Id \to \mathbb{N}$, or $Id \to \Pi$. It is thus convenient to define iterators that can handle functions f of type $(f \in) Id \to A$, for any discretely ordered domain A. Let thus A be discretely ordered; we define $iter_A : (Id \times ((A \times A) \to A) \times (Id \to A) \times A) \to A$ by:

$$iter_A(\iota, op, f, a) = \begin{cases} a, & \text{if } \iota = 0 \\ op(f(\iota - 1), iter_A(\iota - 1, op, f, a)), & \text{if } \iota > 0 \end{cases}$$

For example, we have $iter_A(3, op, f, a) = op(f(2), op(f(1), op(f(0), a)))$. The well-definedness of $iter_A$ follows by induction. Also, continuity of $iter_A$ follows easily when A is discretely ordered[7]. In the sequel, we will employ the following notation which seems more readable:

[7]In our case, A can only be $Bool, \mathbb{N}$ or Π, which are all discretely ordered, and Id is also discretely ordered. It is easy to see that the mappings $iter_A$ are indeed continuous, because:

- if A and B are discretely ordered domains then so are $A \times B$, and $A \to B$ (and $A + B$), and
- if B is any domain, and A is discretely ordered, then any function $f : A \to B$ is continuous.

$$op^{\,\iota}_{a:A}(f) \overset{not.}{=} iter_A(\iota, op, f, a)$$

The mappings $op^{\,\iota}_{a:A}(f)$ provide us with useful abstractions. For example, the predicate *terminates* can be expressed as follows: $terminates(\overline{\iota}, \vartheta) = \bigwedge^{\overline{\iota}}_{true:Bool}(\lambda\iota.isnull(\vartheta(\iota)))$.

We also use these abstract iterators in the definition of the process scheduler function $sched : (Cont \times R) \to \Pi$. $sched(\gamma, \rho)$ computes the number of processes and the number of synchronization pairs in a given continuation $\gamma \in Cont$, and next it uses this information to choose at random - i.e. by using the random number $\rho \in R$ - an element of type Π.

$$\begin{aligned}
sched(\gamma,\rho) = \ & \mathsf{let}\ n_P = |\gamma|^P, n_S = |\gamma|^S\ \mathsf{in} \\
& \mathsf{if}\ ((n_P + n_S) = 0)\ \mathsf{then}\ nil \\
& \mathsf{else}\quad \mathsf{let}\ i = \rho\%(n_P + n_S)\ \mathsf{in} \\
& \qquad\qquad \mathsf{if}\ (i < n_P)\ \mathsf{then}\ ith_P(\gamma, i)\ \mathsf{else}\ ith_S(\gamma, i - n_P)
\end{aligned}$$

$|\cdot|^P, |\cdot|^S : Cont \to \mathbb{N}$ are cardinal computing functions defined as follows:

$$\begin{aligned}
|(\overline{\iota}, \vartheta)|^P &= +^{\overline{\iota}}_{0:\mathbb{N}}(\lambda\iota.z(proc(\iota, \vartheta))) \\
|(\overline{\iota}, \vartheta)|^S &= +^{\overline{\iota}}_{0:\mathbb{N}}(\lambda\iota_1. +^{\overline{\iota}}_{0:\mathbb{N}}(\lambda\iota_2.z(sync(\iota_1, \iota_2, \vartheta))))
\end{aligned}$$

where $proc : (Id \times Kont) \to \Pi$ and $sync : (Id \times Id \times Kont) \to \Pi$ are:

$$proc(\iota, \vartheta) = \begin{cases} \iota, & \mathsf{if}\ \vartheta(\iota) = (proc, \phi) \\ nil, & \text{otherwise} \end{cases}$$

$$sync(\iota_1, \iota_2, \vartheta) =$$

$$\begin{cases} (\iota_1, \iota_2, \xi, v), & \mathsf{if}\ \vartheta(\iota_1) = (c_1!\xi, \phi_1), \vartheta(\iota_2) = (c_2?v, \phi_2),\ \text{and}\ c_1 = c_2 \\ nil, & \text{otherwise} \end{cases}$$

and $z : \Pi \to \mathbb{N}$ is defined by: $z(\pi) = \ \mathsf{if}\ \pi = nil\ \mathsf{then}\ 0\ \mathsf{else}\ 1$.

One can easily check that $proc, sync$ and $isnull$ are monotone mappings[8]. By using the fact that, if A is any domain and B is discretely ordered then any monotone function $f : A \to B$ is continuous, it follows

[8]To show this one uses the fact that, if $(\varsigma_1, \phi_1) \sqsubseteq (\varsigma_2, \phi_2)$ then $\varsigma_1 = \varsigma_2$ (for every $(\varsigma_1, \phi_1), (\varsigma_2, \phi_2) \in Proc$); this is so because $\varsigma_1, \varsigma_2 \in Sched$ and $Sched$ is discretely ordered.

that $proc, sync$ and $isnull$ are continuous mappings. $z : \Pi \to \mathbb{N}$ is also continuous; in this case it suffices to see that Π is discretely ordered.

The mapping $ith_P : (Cont \times \mathbb{N}) \to \Pi$ searches throughout a space of processes, and the mapping $ith_S : (Cont \times \mathbb{N}) \to \Pi$ searches throughout a space of pairs of processes that can synchronize. Both $ith_P(\gamma, i)$ and $ith_S(\gamma, i)$ return a reference to the element that is on the i'th position in the corresponding search space. The search is performed with respect to the natural ordering on $Id = \mathbb{N}$, respectively with respect to (a relationship that is derived from) the lexical order on $Id \times Id$. The definitions for ith_P and ith_S are given below.

$$ith_P((\bar{\iota}, \vartheta), i) =$$

$$\bigoplus_{nil:\Pi}^{\bar{\iota}}(\lambda\iota.\ \mathsf{if}\ (pos_P(\iota, (\bar{\iota}, \vartheta)) = i)\ \mathsf{then}\ proc(\iota, \vartheta)\ \mathsf{else}\ nil)$$

$$ith_S((\bar{\iota}, \vartheta), i) =$$

$$\bigoplus_{nil:\Pi}^{\bar{\iota}}(\lambda\iota_1.\bigoplus_{nil:\Pi}^{\bar{\iota}}(\lambda\iota_2.\ \mathsf{if}\ (pos_S(\iota_1, \iota_2, (\bar{\iota}, \vartheta)) = i)\ \mathsf{then}\ sync(\iota_1, \iota_2, \vartheta)$$

$$\mathsf{else}\ nil))$$

The auxiliary binary operator $\oplus : (\Pi \times \Pi) \to \Pi$ simply helps in selecting the first non-*nil* element in a sequence: $\oplus(\pi_1, \pi_2) = \ \mathsf{if}\ \pi_1 = nil\ \mathsf{then}\ \pi_2\ \mathsf{else}\ \pi_1$.

Finally, for any continuation γ, the operator $pos_P(\iota, \gamma)$ determines the *position* of the process with identifier ι, and the operator $pos_S(\iota_1, \iota_2, \gamma)$ determines the *position* of a pair of processes that can synchronize and have identifiers ι_1 and ι_2. In the first case, the position of the process with identifier ι is determined with respect to the natural ordering on $Id = \mathbb{N}$, and we compute it by counting the number of processes with identifiers $\iota' < \iota$. The definition of the auxiliary mapping $pos_P : (Id \times Cont) \to \mathbb{N}$ is as follows:

$$pos_P(\iota, (\bar{\iota}, \vartheta)) = \sum_{0:\mathbb{N}}^{\iota}(\lambda\iota'.z(proc(\iota', \vartheta)))$$

For the definition of $pos_S : (Id \times Id \times Cont) \to Id$ we consider the lexical order. According to the lexical order, $(\iota'_1, \iota'_2) < (\iota_1, \iota_2)$ if either $\iota'_1 = \iota_1$ and $\iota'_2 < \iota_2$, or $\iota'_1 < \iota_1$. But, for a given continuation $\gamma = (\bar{\iota}, \vartheta)$, we only consider pairs (ι_1, ι_2) such that $\iota_1 < \bar{\iota}$ and $\iota_2 < \bar{\iota}$; this means that, we have to consider a relationship over $Id \times Id$ which (for convenience we also denote by '<', and which) is defined as follows: $(\iota'_1, \iota'_2) < (\iota_1, \iota_2)$ if either $\iota'_1 = \iota_1$ and $\iota'_2 < \iota_2$, or $\iota'_1 < \iota_1$ in which case we must have $\iota'_2 < \bar{\iota}$. The definition of pos_S is given by:

$$
\begin{aligned}
&pos_S(\iota_1, \iota_2, (\bar{\iota}, \vartheta)) = \\
&\qquad \Big(+^{\iota_2}_{0:\mathbb{N}} (\lambda \iota_2'.z(sync(\iota_1, \iota_2', \vartheta))) \Big) \\
&\qquad + \ \Big(+^{\iota_1}_{0:\mathbb{N}} (\lambda \iota_1'. +^{\bar{\iota}}_{0:\mathbb{N}} (\lambda \iota_2'.z(sync(\iota_1', \iota_2', \vartheta)))) \Big)
\end{aligned}
$$

4 Solving our domain equation

In sections 2 and 3 we have employed domains defined by equations of the following form:

$$D = (E_1 \times (E_1 \to (E_2 \times D))) \to E,$$

where E is a domain with least element[9], and E_1, E_2 are arbitrary domains.

In solving this equation we follow the approach (and use the notation) in [12]. We can define $D_0 = \emptyset$ and put $D_{i+1} = (E_1 \times (E_1 \to (E_2 \times D_i))) \to E$. We want to build a co-limiting cone of domains and embeddings, but we need to ensure that the corresponding projections are total. Obviously, there is a unique embedding of D_0 into D_1, but the projection corresponding to this embedding is not total. However, we note that D_1, which is $D_1 = \emptyset \to E$, has exactly one element: the mapping with empty graph; we denote this element by d_0. Next, we consider D_2 which is: $D_2 = (E_1 \times (E_1 \to (E_2 \times D_1))) \to E$. We can define the embedding $f_1 : D_1 \lhd D_2$ as follows: $f_1(d_0) = \lambda\gamma.\bot_E \ (= \bot_{D_2})$, where γ ranges over $E_1 \times (E_1 \to (E_2 \times D_1))$. The corresponding projection ($f_1^P(d) = d_0$ for any $d \in D_2$) is total. By using the framework in [12] we infer that

$$f_{i+1} = (\mathrm{id}_{E_1} \times (\mathrm{id}_{E_1} \overset{E}{\to} (\mathrm{id}_{E_2} \times f_i))) \overset{E}{\to} \mathrm{id}_E$$

is a good definition for all $i > 0$. Indeed, $\mathrm{id}_{E_1}, \mathrm{id}_{E_2}$ and id_E are identity functions, and one can easily check by induction that the projection f_i^P corresponding to f_i is total for any $i \in \omega$ (f_1^P is total, and from the fact that f_i^P is total it follows that f_{i+1}^P is also total).

We get the co-limiting cone from the ω-sequence of domains and embeddings

$$D_1 \overset{f_1}{\lhd} D_2 \overset{f_2}{\lhd} \cdots \overset{f_{i-1}}{\lhd} D_i \overset{f_i}{\lhd} D_{i+1} \overset{f_{i+1}}{\lhd} \cdots$$

as in [12] and obtain a domain D such that: $D \cong (E_1 \times (E_1 \to (E_2 \times D))) \to E$.

[9]In the equations given in sections 2 and 3, $E = R \to State \to O$, and $\bot_E = \lambda\rho.\lambda\sigma.\bot_O$.

5 Concluding remarks and future work

The work given in this paper suggests that, by employing the CSC technique [13, 16], denotational semantics can be used as a general method for concurrent languages prototyping. We plan to validate this idea by employing the CSC technique in the development of denotational prototype models for more complex concurrent languages, including POOL [1], and Concurrent Idealized Algol [3]. For the CSC technique, we also plan to study whether there exists a formal relationship between denotational models that yield elements of powerdomain constructions and corresponding denotational (prototype) models that yield single arbitrary execution traces. Such a theoretical study could be accomplished both within the classic domain theory [11] and within the mathematical framework of complete metric spaces [2].

References

[1] **America, P.**: Issues in the design of a parallel object-oriented language, *Formal Aspects of Computing*, 1:366-411, 1989.

[2] **De Bakker, J.W., De Vink, E.P.**: *Control flow semantics*, MIT Press, 1996.

[3] **Brookes, S.**: The essence of parallel Algol, *Information and Computation*, 179(1):118-149,2002.

[4] Haskell home page, `http://www.haskell.org/`

[5] **Hieb, R., Dybvig, R.K. , Anderson, C.W.**: Subcontinuations, *Lisp and Symbolic Computation*, 7(1):83-110, 1994.

[6] **Hoare, C.A.R.**: Communicating Sequential Processes, *Communications of the ACM, 21:667–677*, 1978.

[7] **Hoare, C.A.R.**: *Communicating Sequential Processes*, Prentice-Hall, 1985.

[8] **INMOS Ltd.**: *Occam programming manual*, Prentice-Hall, 1984.

[9] **Mosses, P.D.**: *Mathematical semantics and compiler generation*, Ph.D. thesis, Univ. Oxford, 1975.

[10] **Mosses, P.D.**: SIS, Semantics Implementation System: Reference manual and user guide, Tech. monograph MD-30, Computer Science Dept., Aarhus Univ., 1979.

[11] **Plotkin, G.D.**: *The category of complete partial orders: a tool for making meanings*, Lecture notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa, June 1978.

[12] **Tennent, R.D.**: *Semantics of Programming Languages*, Prentice-Hall, 1991.

[13] **Todoran, E.**: Metric semantics for synchronous and asynchronous communication: a continuation-based approach, In *Proc. of FCT'99 Workshop on Distributed Systems*, *Electronic Notes in Theoretical Computer Science (ENTCS)*, 28:119–146, Elsevier, 2000.

[14] **Todoran, E.**: *Semantic techniques in concurrent languages development*, Ph.D. thesis, Technical University of Cluj-Napoca, Romania, 2000.

[15] **Todoran, E.**: Denotational interpreter for a CSP-like language, *Automation, Computers, Applied Mathematics*, 11(2):19-33, 2002.

[16] **Todoran, E., Papaspyrou, N.**: Continuations for parallel logic programming, In *Proc. of 2nd International ACM-SIGPLAN Conference on Principles and practice of Declarative Programming (PPDP'00)*, pages 257–267, 2000.

[17] **Wand, M.**: Continuation-based multiprocessing, In *Conference record of the 1980 Lisp Conference*, pages 19-28, 1980.