Denotational prototype semantics for a simple CSP-like language

Eneia Todoran[1] Nikolaos Papaspyrou[2] Kalman Pusztai[1]

[1]Technical University of Cluj-Napoca
department of computer science
Baritiu Street, 28, Cluj-Napoca
Romania
{Eneia.Todoran,Kalman.Pusztai}@cs.utcluj.ro

[2]National Technical University of Athens
Department of Electrical and Computer Engineering
Polytechnioupoly, 15780 Zografou, Athens
Greece
nickie@softlab.ntua.gr

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 method for concurrent languages prototyping. 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 in developing a denotational prototype semantics for a simple concurrent language providing constructs for CSP-like synchronous communication. The CSC technique plays the main role in the design of the denotational model.**

I. INTRODUCTION

In software engineering, a prototype is an initial version of a system which is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions. Ideally, a prototype serves as a mechanism for identifying software requirements. Rapid development of the prototype is essential so that costs are controlled and users can experiment with the prototype early in the software process.

Denotational semantics is a well-known method for formal specification and design of computer languages; its main characteristic is *compositionality*. It is easy to use a functional language and classic denotational techniques to produce rapidly compositional prototypes for various (aspects of) sequential programming languages (we only mention here the early work of Peter Mosses on the use of denotational descriptions in compiler generation [7, 8]. However, to the best of our knowledge, denotational semantics have never been used systematically as a prototyping method for concurrent languages, and all our attempts to get a satisfactory solution to this problem by using only classic compositional techniques have failed.

This paper shows that, by using the "continuation semantics for concurrency" (CSC) technique - recently introduced by us [13, 14] - denotational semantics can be used not only as a method for formal specification and design, but also as a 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[1]. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. We call such a denotational model a *denotational prototype*. The immediate implementation of such a denotational model in an appropriate functional language (such as Haskell [10]) is a compositional interpreter for the concurrent language under study.

The CSC technique was introduced in [13] using metric semantics [2]. In this paper we employ classic (cpo-based) domains and continuous functions in developing a denotational prototype semantics for a simple concurrent language, providing constructs for parallel composition and CSP-like synchronous communication [4, 5]; the CSC technique plays the main role in the semantic design. We emphasize that, when the CSC technique is used 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, 6].

II. SYNTAX AND INFORMAL EXPLANATION

We consider a simple CSP-like language, called L_{CSP}. The syntax of L_{CSP} is given below in BNF. We assume given a set $(v \in)Var$[2] of (numerical) variables, a set $(e \in)Exp$ of numerical expressions, a set $(b \in)BExp$ of boolean expressions, a set $(c \in)Chan$ of communication channels, and a set $(x \in)PVar$ of procedure variables (or procedure identifiers).

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

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

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

[1]As shown in [13], the CSC technique can be used without difficulty to produce elements of appropriate powerdomain constructions, but this is not the subject of the present paper.

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

We assume that numerical and boolean expressions have no side effects and their evaluation always terminates. For simplicity, variables are only of numerical type (however, variables can appear in boolean expressions such as: $v_1 < v_2 + 10$).

L_{CSP} provides assignment ($v := e$), a primitive for writing the value of a numerical expression at the standard output file (**write**(e)), two constructs for synchronous communication ($c!e$ and $c?v$), 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 constructs $c!e$ and $c?v$ are as in Occam [9]. 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.

III. SEMANTICS

Numerical expressions evaluate to natural numbers ($\in \mathbb{N}$), and boolean expressions evaluate to boolean values ($\in Bool = \{true, false\}$). The meaning of expressions is defined with respect to a given domain $State$ of states:

$$(\sigma \in) State = Var \to \mathbb{N}.$$

The following valuations are assumed given:

$$\mathcal{E}[\![\cdot]\!] : Exp \to (State \to \mathbb{N}),$$

$$\mathcal{B}[\![\cdot]\!] : BExp \to (State \to Bool)$$

and we let ξ range over ($\xi \in$) $State \to \mathbb{N}$.

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:

$$O \cong (\{\epsilon\} + \{\delta\} + (\mathbb{N} \times O))_\perp,$$

where ϵ is the empty sequence, and δ is a constant that denotes *deadlock*. O is a lifted domain; the bottom element represents a non-terminating computation that produces no observable effect.

To simulate the nondeterministic behavior of a "real" 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 ρ_0($\in R$) and a mapping[3] $r : R \to R$ that produces a new random number from a given one.

The following domain is used for the denotational semantics:

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

where $Cont$ is the domain of *continuations* which, in the CSC approach, is a configuration (in simple applications a multiset) of computations (i.e. of elements of type D). Here, we implement this concept as follows:

[3] A very simple example of such a pseudo-random number generator can be defined as follows:

$$r(\rho) = (25173 \times \rho + 13849)\,\%\,65536,$$

where $\rho_0 = 17489$.

$$(\gamma \in) Cont = Id \times Kont$$

$$(\vartheta \in) Kont = Id \to Proc$$

$$(p \in) Proc = Sched \times D$$

The elements of the domain Id are *process identifiers*; they are auxiliary entities that can be used in the representation of continuations in the CSC approach. For an appropriate semantic modeling Id should contain an infinite number of elements. 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 some scheduling information and a computation (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.

The domain $Sched$ is defined as follows:

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

where

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

$$Rcv = Chan \times Var.$$

For easier readability, we denote typical elements (c, ξ) of Snd by $c!\xi$, and we denote typical elements (c, v) of Rcv by $c?v$.

Whenever convenient, for any continuation $\gamma = (\bar{\iota}, \vartheta) \in Cont$ we will use the following abbreviations:

- $\gamma[\iota] \stackrel{not.}{=} \vartheta(\iota)$
- $\gamma\langle\iota\rangle \stackrel{not.}{=}$ let $(\varsigma, \phi) = \vartheta(\iota)$ in ϕ
- $\langle\gamma \mid \iota_1 \mapsto p_1 \mid \ldots \mid \iota_n \mapsto p_n\rangle \stackrel{not.}{=}$

  $(\bar{\iota}, (\vartheta \mid \iota_1 \mapsto p_1 \mid \ldots \mid \iota_n \mapsto p_n))$

We see that, in order to get a good mathematical foundation for our design, we need to solve the following domain equation (in which 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 ($\perp_E = \lambda\rho.\lambda\sigma.\perp_O$), which means that D also has a least element; more precisely $\perp_D = \lambda\gamma.\perp_E$.

Finally, to deal with recursion we define *semantic environments* as follows: ($\eta \in$)$Env = PVar \to D$.

We emphasize that, apart from $O, D, Proc, Cont, Kont$ and Env, all domains that are employed in the semantic constructions given in sections III and IV are discretely ordered.

Continuations play a central role in the CSC approach [13, 14]. Moreover, in this paper we show that the information contained in continuations suffices for all process scheduling purposes, including process synchronization, termination and deadlock detection. In the sequel, we only present a particular solution to the problem of designing such a "pure" continuation-based approach to communication and concurrency using classic domains and continuous functions.

We begin by defining the predicate $terminates : Cont \to Bool$ that formalizes the intuitive notion of *termination*. We put $terminates(0, \vartheta) = true$, and, when $0 < \bar{\iota}$:

$$terminates(\bar{\iota}, \vartheta) = \bigwedge_{0 \le \iota < \bar{\iota}} isnull(\vartheta(\iota)),$$

where the predicate $isnull : Proc \to Bool$ is given by:

$$isnull(\varsigma, \phi) = (\varsigma = null).$$

It is easy to prove the continuity of $terminates$, but we defer the issue to section IV.

For scheduling purposes, it is also convenient to introduce the following 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 chooses at random a process identifier $\iota \in Id$ such that $\gamma[\iota] = (proc, \phi)$ for some $\phi \in D$, or
- 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,
- when none of the above choices are possible, it returns nil, which signifies *deadlock* detection.

Section IV offers an example of such a function $sched$.

We are now prepared for the definition of the denotational semantics.

Definition 2 *(Denotational prototype semantics for L_{CSP})*

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

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

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

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

where we have used the following notations: $\widetilde{(\varsigma, \phi)} \stackrel{not.}{=} (proc, \phi)$, $\widetilde{(\varsigma, \phi)} \stackrel{not.}{=} (null, \phi)$, and $\sigma' = (\sigma \mid v \mapsto \xi\,\sigma)$.

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

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

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

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

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

$$[\![\,c?v\,;\,s\,]\!]\eta\gamma\rho\sigma = \kappa((c?v, [\![\,s\,]\!]\eta) :: \gamma)\rho\sigma$$

$$[\![\,\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\,]\!]\eta\gamma\rho\sigma = \begin{cases} [\![\,s_1\,]\!]\eta\gamma\rho\sigma, & \text{if } \mathcal{B}[\![\,b\,]\!]\sigma = true \\ [\![\,s_2\,]\!]\eta\gamma\rho\sigma, & \text{if } \mathcal{B}[\![\,b\,]\!]\sigma = false \end{cases}$$

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

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

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

where % is the modulo *operator and we have used the notation:*

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

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

(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.

Of course, in the definition above fix is the classical fixed point operator[4]. It is not difficult to see that (provided $terminates$ and $sched$ are continuous) K is a continuous mapping and thus κ is well-defined and continuous, and $[\![\cdot]\!]$ is also continuous.

A distinguished feature of our semantic model consists in the fact that, the final yield of the denotational model is not an element of some powerdomain construction but rather a single execution trace. Moreover, 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 an execution trace.

IV. A PROCESS SCHEDULER WITH RANDOM CHOICE

The specification that we gave in section III does not determine a unique function $sched : (Cont \times R) \to \Pi$. In this section we 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 $0 \le \iota < \bar{\iota}$. 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

[4] 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.

$Id \to Bool$, $Id \to \mathbb{N}$, 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 $iter_A : (Id \times ((A \times A) \to A) \times (Id \to A) \times A) \to A$ as follows:

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

The well-definedness of $iter_A$ follows by induction. Also, continuity of $iter_A$ follows easily when A is discretely ordered[5]. This definition gives us:

$$\begin{aligned} iter_A(0, op, f, a) &= a \\ iter_A(1, op, f, a) &= op(f(0), a) \\ iter_A(2, op, f, a) &= op(f(1), op(f(0), a)) \end{aligned}$$

and so on. In the sequel, we will employ the following notation which seems often more readable:

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

The mappings $op^{\iota}_{a:A}(f)$ provide us with useful abstractions. For example, the predicate *terminates* (introduced together with the predicate *isnull* in section III) can be expressed as follows:

$$terminates(\bar{\iota}, \vartheta) = \bigwedge^{\bar{\iota}}_{true:Bool}(\lambda\iota.isnull(\vartheta(\iota))).$$

We also use these 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.

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

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

$$|(\bar{\iota}, \vartheta)|^P = \mathbf{+}^{\bar{\iota}}_{0:\mathbb{N}}(\lambda\iota.z(proc(\iota, \vartheta)))$$

$$|(\bar{\iota}, \vartheta)|^S = \mathbf{+}^{\bar{\iota}}_{0:\mathbb{N}}(\lambda\iota_1.\mathbf{+}^{\bar{\iota}}_{0:\mathbb{N}}(\lambda\iota_2.z(sync(\iota_1, \iota_2, \vartheta))))$$

where the mappings $proc : (Id \times Kont) \to \Pi$ and $sync : (Id \times Id \times Kont) \to \Pi$ are given by:

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

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

and $z : \Pi \to \mathbb{N}$ is:

$$z(\pi) = \begin{cases} 0, & \text{if } \pi = nil \\ 1, & \text{otherwise} \end{cases}$$

One can easily check that *proc*, *sync* and *isnull* (*isnull* was introduced in section III) are monotone mappings[6]. 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 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 (more precisely an element of type Π) 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.

$$ith_P((\bar{\iota}, \vartheta), i) = \bigoplus^{\bar{\iota}}_{nil:\Pi}(\lambda\iota.\pi_P)$$

$$ith_S((\bar{\iota}, \vartheta), i) = \bigoplus^{\bar{\iota}}_{nil:\Pi}(\lambda\iota_1.\bigoplus^{\bar{\iota}}_{nil:\Pi}(\lambda\iota_2.\pi_S))$$

where we used the following abbreviations:

$$\begin{aligned} \pi_P = {} & \text{if } (pos_P(\iota, (\bar{\iota}, \vartheta)) = i) \\ & \text{then } proc(\iota, \vartheta) \\ & \text{else } nil \end{aligned}$$

and

$$\begin{aligned} \pi_S = {} & \text{if } (pos_S(\iota_1, \iota_2, (\bar{\iota}, \vartheta)) = i) \\ & \text{then } sync(\iota_1, \iota_2, \vartheta) \\ & \text{else } nil \end{aligned}$$

The auxiliary binary operator $\oplus : (\Pi \times \Pi) \to \Pi$ simply helps in selecting the first non-*nil* element in a sequence:

$$\oplus(\pi_1, \pi_2) = \begin{cases} \pi_2, & \text{if } \pi_1 = nil \\ \pi_1, & \text{otherwise} \end{cases}$$

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:

[5] 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.

[6] 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.

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

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:

$$pos_S(\iota_1,\iota_2,(\bar{\iota},\vartheta)) =$$
$$\left(\textstyle\sum^{\iota_2}_{0:\mathbb{N}}(\lambda\iota_2'.z(sync(\iota_1,\iota_2',\vartheta)))\right) +$$
$$\left(\textstyle\sum^{\iota_1}_{0:\mathbb{N}}(\lambda\iota_1'.\sum^{\bar{\iota}}_{0:\mathbb{N}}(\lambda\iota_2'.z(sync(\iota_1',\iota_2',\vartheta))))\right)$$

V. SOLVING OUR DOMAIN ECUATION

We show how to solve an equation of the form:

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

where E is a domain with least element[7], 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 we put

$$D_{i+1} = (E_1 \times (E_1 \to (E_2 \times D_i))) \to E.$$

Following [12], 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:

$$f_1(d_0) = \lambda\gamma.\bot_E \qquad (= \bot_{D_2})$$

where γ ranges over $(\gamma \in) E_1 \times (E_1 \to (E_2 \times D_1))$. The corresponding projection

$$f_1^P(d) = d_0 \qquad \text{(for any } d \in D_2)$$

is total. By using the framework in [12] we infer that

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

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 can then construct 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

$$D \cong (E_1 \times (E_1 \to (E_2 \times D))) \to E.$$

VI. CONCLUDING REMARKS AND FUTURE WORK

In this paper, we have presented a denotational prototype semantics for a simple CSP-like language. The denotational model was designed by using the CSC technique [13, 14]. We worked in a category of complete partial orders and continuous mappings [11, 12, 6], and we used the notation in [12]. We showed that, 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. This gives rise to a "pure" continuation-based approach to communication and concurrency in which all scheduling tasks can be modeled as operations manipulating continuations.

The work given in this paper suggests that, by employing the CSC technique 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].

VII. REFERENCES

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

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

[3] S. Brookes, "The essence of parallel Algol," *Information and Computation*, vol. 179, no. 1, 2002, pp. 118-149.

[4] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, 1978, pp. 667–677.

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

[6] J.C. Mitchell, *Foundations for Programming Languages*, MIT Press, 1996.

[7] In the original equation $E = R \to State \to O$. The least element of this domain is $\lambda\rho.\lambda\sigma.\bot_O$.

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

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

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

[10] S. Peyton Jones and J. Hughes (editors), *Report on the programming language Haskell 98: a non-strict purely functional language*, Yale Univ., Tech. Report YALEU/DCS/RR-1106, 1999.

[11] G.D. Plotkin, *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, 1978.

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

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

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