Continuation Semantics for Maximal Parallelism and Imperative Programming

Eneia Nicoloae Todoran
Department of Computer Science
Technical University of Cluj-Napoca
Baritiu Street 28, 400027, Cluj-Napoca, Romania
Email: eneia.todoran@cs.utcluj.ro

***Abstract*—We present a denotational semantics for a simple concurrent imperative programming language in which the semantic operator for parallel composition is designed according to the maximal parallelism model of non-interleaved computations. The denotational semantics is designed with metric spaces and continuation semantics for concurrency. We also present a Haskell implementation of our denotational semantics in the form of a prototype interpreter.**

I. Introduction

In previous work we introduced a *continuation semantics for concurrency* (CSC) [15], [6] that can be used to model both sequential and parallel composition in interleaving semantics while providing the general advantages of the technique of continuations [14]. More recently, we investigated the possibility to express synchronization between two or multiple parties in continuation semantics [18], [7]; in [7], [18] we used CSC for this purpose. In [20] we used CSC in combination with classic continuation-passing style to express synchronization between multiple parallel components in a compositional manner.

In this paper we present a denotational semantics for a simple concurrent imperative programming language in which the semantic operator for parallel composition is designed according to the maximal parallelism model of non-interleaved computations. The denotational semantics is designed with metric spaces and continuations for concurrency following the approach introduced in [20]. We also present a Haskell implementation of our denotational semantics in the form of a prototype interpreter.

As it is well known, a number of mathematical theories have been developed based on models of the so-called *true concurrency* (or non-interlaving) kind. The notion of *Petri net* is of fundamental importance in this area. There is a well-known treatment of maximal parallelism in direct semantics; see, e.g., [2] section 15.2. In metric semantics the non-interleaving model was also investigated based on the *pomset* model [3]. Continuation-passing style was developed initially as a tool for denotational semantics [13], [12]. Our present aim is to show that the concept of maximal parallelism can also be expressed in continuation semantics.

A. Contribution

We show that the continuation-based technique introduced in [20] can be used to model *maximal parallelism* (sometimes called *synchronous parallelism*) in a compositional manner. In [7], [18] the semantics of synchronization is expressed by using silent steps or hiatons (see, e.g., [2], chapter 9), which are needed to achieve the contractiveness of some higher-order mappings. The technique introduced in [20] can be used to express synchronization between two or multiple parallel components without using silent steps or hiatons. We present a denotational model for maximal parallelism designed by using the continuation-based technique introduced in [20]. Our denotational (mathematical) semantics is given in Section IV. In Section V we also present a Haskell implementation of our denotational semantics in the form of a prototype interpreter for the language under investigation, that can be easily tested and evaluated.

II. Preliminaries

The notation $(x \in)X$ introduces the set X with typical element x ranging over X. Let $f \in X \to Y$ be a function. The function $(f \mid x \mapsto y) : X \to Y$, is defined (for $x, x' \in X, y \in Y$) by: $(f \mid x \mapsto y)(x') =$ if $x'=x$ then y else $f(x')$. Instead of $((f \mid x_1 \mapsto y_1) \mid x_2 \mapsto y_2)$ we write $(f \mid x_1 \mapsto y_1 \mid x_2 \mapsto y_2)$. In general, instead of $((f \mid x_1 \mapsto y_1) \cdots \mid x_n \mapsto y_n)$ we write $(f \mid x_1 \mapsto y_1 \mid \cdots \mid x_n \mapsto y_n)$. If $f : X \to X$ and $f(x) = x$ we call x a *fixed point* of f. When this fixed point is unique (see Theorem 2.1) we write $x = fix(f)$.

A *multiset* is a generalization of a set. Intuitively, a multiset is a collection in which an element may occur more than once, or an unordered list. One can represent the concept of a multiset of elements of type A by using functions from $A \to \mathbb{N}$, or partial functions from $A \to \mathbb{N}^+$, where $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ ($\mathbb{N}^+$ is the set of natural numbers without 0). Let $(a \in)A$ be a countable set. We use the notation:

$$[A] \stackrel{not.}{=} \bigcup_{X \in \mathcal{P}_{finite}(A)} \{m \mid m \in (X \to \mathbb{N}^+)\}$$

where $\mathcal{P}_{finite}(A)$ is the power set of all *finite* subsets of A. As A is countable, $\mathcal{P}_{finite}(A)$ is also countable. An element $m \in [A]$ is a (finite) multiset of elements of type A, a function $m : X \to \mathbb{N}^+$, for some finite subset $X \subseteq A$, such that $\forall a \in X : m(a) > 0$. $m(a)$ is called the *multiplicity* (number of occurrences) of a in m. $[A]$ is the set of all finite multisets of elements of type A.

One can define various operations on multisets $m_1, m_2 \in [A]$. Below, $\mathsf{dom}(\cdot)$ is the domain of function '$\cdot$'. The *multiset sum* operation $\uplus : ([A] \times [A]) \to [A]$ can be defined as follows:

$$\mathsf{dom}(m_1 \uplus m_2) = \mathsf{dom}(m_1) \cup \mathsf{dom}(m_2)$$

$$(m_1 \uplus m_2)(a) = \begin{cases} m_1(a) + m_2(a) & \text{if} \quad a \in \mathsf{dom}(m_1) \cap \mathsf{dom}(m_2) \\ m_1(a) & \text{if} \quad a \in \mathsf{dom}(m_1) \setminus \mathsf{dom}(m_2) \\ m_2(a) & \text{if} \quad a \in \mathsf{dom}(m_2) \setminus \mathsf{dom}(m_1) \end{cases}$$

When $(a \in)A$ is a countable set we use the following notation convention. We denote by $\overline{a}$ typical elements of $[A]$, i.e., $\overline{a} \in [A]$ is a multiset of elements of the type A.

A. Metric spaces

The denotational semantics given in this paper is built within the mathematical framework of *1-bounded complete metric spaces*. We work with the following notions which we assume known: *metric* and *ultrametric* space, *isometry* (distance preserving bijection between metric spaces, denoted by '$\cong$'), *complete* metric space, and *compact* set. For details the reader may consult, e.g., the monograph [2].

We recall that if (X, d_X), (Y, d_Y) are metric spaces, a function $f{:}X \to Y$ is a *contraction* if $\exists c \in \mathbb{R},\ 0 \leq c < 1$, $\forall x_1, x_2 \in X : d_Y(f(x_1), f(x_2)) \leq c \cdot d_X(x_1, x_2)$. In metric semantics it is customary to attach a contracting factor of $c = \frac{1}{2}$ to each computation step. When $c = 1$ the function f is called *non-expansive*. In what follows we denote the set of all nonexpansive functions from X to Y by $X \stackrel{1}{\to} Y$. The following theorem is at the core of metric semantics.

Theorem 2.1 (Banach): Let (X, d_X) be a complete metric space. Each contraction $f : X \to X$ has a *unique* fixed point.

Let $(a, b \in)A$ is any nonempty set. One can define the *discrete metric* on A ($d : A \times A \to [0, 1]$) as follows: $d(a, b) = 0$ if $a = b$, and $d(a, b) = 1$ otherwise. (A, d) is a complete ultrametric space.

Let A be a nonempty set. Let $(x, y \in)A^\infty = A^* \cup A^\omega$, where $A^*(A^\omega)$ is the set of all finite (infinite) sequences over A. One can define a metric over A^∞ as follows: $d_B(x, y) = 2^{-\sup\{\, n \mid x[n]=y[n] \,\}}$, where $x[n]$ denotes the prefix of x of length n, in case $length(x) \geq n$, and x otherwise (by convention, $2^{-\infty} = 0$). d_B is a Baire-like metric. (A^∞, d_B) is a complete ultrametric space.

Other composed metric spaces can be built up using the composite metrics given in Definition 2.2.

Definition 2.2: Let $(X, d_X), (Y, d_Y)$ be (ultra) metric spaces. On $(x \in)X$, $(f \in)X \to Y$ (the function space), $(x, y) \in X \times Y$ (the Cartesian product), $u, v \in X + Y$ (the disjoint union of X and Y, $X + Y = (\{1\} \times X) \cup (\{2\} \times Y)$), and $U, V \in \mathcal{P}(X)$ (the power set of X) one can define the following metrics:

(a) $d_{\frac{1}{2}\cdot X} : X \times X \to [0, 1]$, $\quad d_{\frac{1}{2}\cdot X}(x_1, x_2) = \frac{1}{2} \cdot d_X(x_1, x_2)$

(b) $d_{X \to Y} : (X \to Y) \times (X \to Y) \to [0, 1]$
$d_{X \to Y}(f_1, f_2) = \sup_{x \in X} d_Y(f_1(x), f_2(x))$

(c) $d_{X \times Y} : (X \times Y) \times (X \times Y) \to [0, 1]$
$d_{X \times Y}((x_1, y_1), (x_2, y_2)) = max\{d_X(x_1, x_2), d_Y(y_1, y_2)\}$

(d) $d_{X+Y} : (X + Y) \times (X + Y) \to [0, 1]$
$d_{X+Y}(u, v) =$
$\quad$ if $(u, v \in X)$ then $d_X(u, v)$
$\quad$ else if $(u, v \in Y)$ then $d_Y(u, v)$ else 1

(e) $d_H : \mathcal{P}(X) \times \mathcal{P}(X) \to [0, 1]$,
$d_H(U, V) = max\{\sup_{u \in U} d(u, V), \sup_{v \in V} d(v, U)\}$
where $d(u, W) = \inf_{w \in W} d(u, w)$ and by convention $\sup \emptyset = 0$, $\inf \emptyset = 1$.

We use the abbreviation $\mathcal{P}_{nco}(\cdot)$ to denote the power set of *non-empty and compact* subsets of '$\cdot$'. Also, we often suppress the metrics part in domain definitions, and write, e.g., $\frac{1}{2} \cdot X$ instead of $(X, d_{\frac{1}{2}\cdot X})$.

Remark 2.3: Let $(X, d_X), (Y, d_Y), d_{\frac{1}{2}\cdot X}$, $d_{X \to Y}, d_{X \times Y}$, d_{X+Y} and d_H be as in Definition 2.2. In case d_X, d_Y are ultrametrics, so are $d_{\frac{1}{2}\cdot X}, d_{X \to Y}, d_{X \times Y}, d_{X+Y}$ and d_H. Moreover, if $(X, d_X), (Y, d_Y)$ are complete then $\frac{1}{2} \cdot X$, $X \to Y$, $X \stackrel{1}{\to} Y$, $X \times Y$, $X + Y$, and $\mathcal{P}_{\mathrm{nco}}(X)$ (with the metrics defined above) are also complete metric spaces [2]. d_H is the *Hausdorff* metric.

III. Syntax of $\mathcal{L}$

We assume given a set $(v \in)V$ of *variables*, a set $(e \in)Exp$ of *expressions*, a set $(b \in)Bexp$ of *boolean expressions* and a set $(y \in)Y$ of *procedure variables*. We assume that the evaluation of an expression always terminates (without producing side effects) and yields an integer value $z \in \mathbb{Z}$. Similarly, the evaluation of a boolean expression yields a boolean value.

Definition 3.1: (Syntax of $\mathcal{L}$) Let $a(\in A) ::= v := e$. A is the set of (elementary) assignment statements. We define the sets of *statements* $(x \in)X$ and *guarded statements* $(g \in)G$ by:

$$\begin{array}{rl} x ::= & \mathsf{skip} \mid a; x \mid \text{ if } b \text{ then } x \text{ else } x \\ & \mid y \mid \text{ letrec } y \text{ be } g \text{ in } x \mid x \parallel x \\ g ::= & \mathsf{skip} \mid a; x \mid \text{ if } b \text{ then } g \text{ else } g \\ & \mid \text{ letrec } y \text{ be } g \text{ in } g \mid g \parallel g \end{array}$$

skip is the inoperative statement. The construction $a; x$ specifies a sequential composition in prefix form. $v := e$ is the assignment statement. Action prefixing is a particular case of sequential composition, which can be encountered, e.g., in CCS or π calculus [9], [10]. The language $\mathcal{L}$ also provides standard constructions for expressing conditional execution (if b then x else x), parallel composition ($x \parallel x$), recursive definitions (letrec y be g in x) and procedure calls (y). Intuitively, the construction letrec y be g in x defines a new procedure y with body g (y may occur recursively in g); notice that the body g of the procedure y is a guarded statement. In a guarded statement each recursive call is preceeded by at least one elementary action, and this guarantees that the semantic operators are contracting functions in the present metric setting [2].

The meaning of expressions is given by a valuation

$$\mathcal{E}[\![\cdot]\!] : Exp \to \Sigma \to \mathbb{Z}$$

where $(\sigma \in)\Sigma = Var \to \mathbb{Z}$ is the set of *states*. Similarly, the meaning of boolean expressions is given by a valuation

$$\mathcal{B}[\![\cdot]\!] : Exp \to \Sigma \to \{ \text{ true }, \text{ false } \}$$

Let $(\alpha \in)Act = V \times (\Sigma \to \mathbb{Z})$. We let ξ range over $\Sigma \to \mathbb{Z}$. Act is the set of partially evaluated (elementary) assignment statements. An element $(v, \xi) \in Act$ is a pair, consisting of a variable v and a partially evaluated expression $\xi (\in \Sigma \to \mathbb{Z})$.

IV. Continuation-based denotational semantics

We present a denotational semantics designed with continuations for the language $\mathcal{L}$. The semantic operator for parallel composition in $\mathcal{L}$ is designed according to the maximal parallelism model of non-interleaved computations

A. Semantic domain

The final yield of our continuation-based semantics is a standard linear-time domain $(p \in)\mathbf{P}$

$$\mathbf{P} = \mathcal{P}_{nco}(\Sigma^* \cup \Sigma^\omega)$$

Here, Σ^* is the collection of all finite (posibly empty) sequences over Σ, and Σ^ω is the collection of all infinite sequences over Σ. We use the symbol ϵ to represent the empty sequence. We view $(q \in)\Sigma^* \cup \Sigma^\omega$ as a complete ultrametric space by endowing it with the Baire metric (see Section II). We use the notation $\sigma \cdot p = \{\sigma \cdot q \mid q \in p\}$, for any $\sigma \in \Sigma$ and $p \in \mathbf{P}$.

In Section IV-C we define a denotational semantics $[\![\cdot]\!]$ for $\mathcal{L}$. The semantic domain of $[\![\cdot]\!]$ is $\mathbf{D}$:

$$(\phi \in)\mathbf{D} \cong \mathbf{F} \stackrel{1}{\to} \Sigma \to \mathbf{P}$$

$$(\kappa \in)\mathbf{K} = \{\kappa_0\} + \tfrac{1}{2} \cdot \mathbf{D}$$
$$(f \in)\mathbf{F} = ([Act] \times \mathbf{K}) \stackrel{1}{\to} \Sigma \to \mathbf{P}$$
$$(\eta \in)\mathbf{Env} = Y \to \mathbf{D}$$

$\mathbf{D}$ is the domain of *denotations*. Following [20], we call $\mathbf{F}$ the domain of *synchronous continuations* and $\mathbf{K}$ is the domain of *asynchronous continuations*. The combination of synchronous and asynchronous continuations can express the non-interleaved execution of parallel components (maximal or synchronous parallelism) in a compositional manner.

Intuitively, a non-empty asynchronous continuation κ stores a $\mathbf{D}$ computation; κ is a parallel composition of computations. In general, such an (asynchronous) continuation is a more complex structure [17], e.g., a tree [7], [15], or a multiset of computations [4], [5], [19]. In the case of $\mathcal{L}$, a continuation is a multiset of computations that are packed into a single computation by means of parallel composition. In this paper, an asynchronous continuation κ is a computation, stored in the space $\mathbf{K} = \{\kappa_0\} + \frac{1}{2} \cdot \mathbf{D}$. κ_0 is the empty continuation. The spaces $\mathbf{D}$ and $\frac{1}{2} \cdot \mathbf{D}$ have the same support set, only the distance between points is halved in $\frac{1}{2} \cdot \mathbf{D}$.

$(\overline{\alpha} \in)[Act]$ is the set of all finite multisets of elements of the type Act. We recall our convention to denote by $\overline{\alpha}$ typical elements of $[Act]$ (when α ranges over Act; see Section II). In the equations given above the sets $[Act]$ and Σ are endowed with the discrete metric which is an ultrametric. Any set endowed with the discrete metric is a complete ultrametric space. An element $\overline{\alpha}$ of the set $[Act]$ is a finite multiset of partially evaluated assignment statements to be executed concurrently (in a non-interleaved manner). The composed metric spaces are built up using the metrics of Definition 2.2. According to [1] the above system of equations has a solution, which is *unique* up to isometry ($\cong$). The solution for $\mathbf{D}$ is obtained as a complete ultrametric space. Finally, $\mathbf{Env}$ is the domain of *semantic environments*, that we use to describe recursive definitions in a compositional manner.

B. Semantic operators in continuation semantics

We define an operator for parallel composition on denotations $\|\colon (\mathbf{D} \times \mathbf{D}) \stackrel{1}{\to} \mathbf{D}$ and an operator for parallel composition on continuations $|||\, : (\mathbf{K} \times \mathbf{K}) \stackrel{1}{\to} \mathbf{K}$ as follows.

$$\kappa_0 ||| \kappa_0 = \kappa_0$$
$$\kappa_0 ||| \phi = \phi$$
$$\phi \;||| \kappa_0 = \phi$$
$$\phi_1 ||| \phi_2 = \phi_1 \parallel \phi_2$$

$$\phi_1 \parallel \phi_2 = \lambda f.\lambda\sigma.((\phi_1 \lfloor \phi_2)(f)(\sigma) \cup (\phi_2 \lfloor \phi_1)(f)(\sigma))$$

The auxiliary operator $\lfloor\colon (\mathbf{D} \times \mathbf{D}) \stackrel{1}{\to} \mathbf{D}$ is given by:

$$\phi_1 \lfloor \phi_2 = \lambda f.\phi_1(\lambda(\overline{\alpha}_1, \kappa_1).\phi_2(\lambda(\overline{\alpha}_2, \kappa_2).f(\overline{\alpha}_1 \uplus \overline{\alpha}_2, \kappa_1 ||| \kappa_2)))$$

Following [16], the semantics of parallel composition is modeled in continuation semantics as a non-deterministic choice between two alternative computations: one starting from the first parallel component and another starting from the second. It is easy to check that the operators $\|, |||$ and $\lfloor$ are non-expansive in both their arguments.

In [4], [5] we showed that it is also possible to describe a computation model based on parallel rewriting of multisets (a form of maximal parallelism) by using a combination of continuation semantics and direct semantics.

C. Denotational semantics

In Definition 4.3 we present the denotational mapping $[\![\cdot]\!] : X \to \mathbf{D}$. The behavior of $[\![\cdot]\!]$ depends on three parameters: a semantic environment, a (synchronous) continuation and a state. In Definition 4.1 we introduce an initial synchronous continuation f_0 as fixed point of a higher order mapping.

Definition 4.1: Let $(\varsigma \in)Act^*$ be the set of all finite, possibly empty, sequences over Act. We denote the empty sequence over Act by ς_ϵ ($\varsigma_\epsilon \in Act^*$).

Let $upd : ([Act] \times \Sigma) \to \mathcal{P}_{finite}(\Sigma)$ be given by:

$$upd\,(\overline{\alpha}, \sigma) = \{\, assign\,(\varsigma, \sigma) \mid \varsigma \in perm\,(\overline{\alpha})\}$$

where we denote by $perm\,(\overline{\alpha})$ the set of all permutations of the multiset $\overline{\alpha} \in [Act]$,[1] and $assign : (Act^* \times \Sigma) \to \Sigma$

$$assign\,(\varsigma_\epsilon, \sigma) = \sigma$$
$$assign\,((v, \xi)\varsigma, \sigma) = assign\,(\varsigma, (\sigma \mid v \mapsto \xi(\sigma)))$$

In the second equation defining $assign$, $(v, \xi)\varsigma$ is the Act^* sequence with head (v, ξ) and tail ς.

Let $\Psi : \mathbf{F} \to \mathbf{F}$ be given by:

$$\Psi(f)(\overline{\alpha}, \kappa)(\sigma) = \bigcup_{\sigma' \in\, upd\,(\overline{\alpha}, \sigma)} (\sigma' \cdot (\ \mathsf{case}\ \kappa\ \mathsf{of}\ \kappa_0 \to \{\epsilon\}; \phi \to \phi(f)(\sigma')))$$

We define $f_0 = fix(\Psi)$.

[1] Obviously, $perm\,(\overline{\alpha}) \in \mathcal{P}_{finite}(Act^*)$.

Remark 4.2: $\Psi : \mathbf{F} \xrightarrow{\frac{1}{2}} \mathbf{F}$, i.e., Ψ is a contraction (hence it has a *unique* fixed point) in particular due to the "$\sigma' \cdot \ldots$" - step in its definition.
Some explanations concerning the definition of f_0 are necessary. If $\kappa = \kappa_0$ (κ_0 is the empty asynchronous continuation) then the execution terminates. If $\kappa = \phi$, for some $\phi \in \mathbf{D}$, then ϕ is evaluated with parameters f_0 and σ', respectively; σ' is computed by the auxiliary mapping upd. upd takes as parameters a multiset $\overline{\alpha}(\in [Act])$ and a state $\sigma(\in \Sigma)$ and yields a finite set of states. The (partially evaluated) assignment statements in the multiset $\overline{\alpha}$ are executed simultaneously if they are independent. If two or more such (partially evaluated) statements share variables their execution order may matter. Each execution order may produce a different effect, hence a different new state. upd first computes the set of all permutations of the multiset $\overline{\alpha}$; each permutation of $\overline{\alpha}$ is a (a sequence or) a list of (partially evaluated) assignment statements corresponding to a particular order of execution of the elements in $\overline{\alpha}$. The effect of the execution of such a list of assignment statements is computed by the mapping $assign$.

Definition 4.3: (Denotational semantics) We define $[\![\cdot]\!] : X \to \mathbf{D}$ by:

$$[\![\text{ skip }]\!](\eta) = \lambda f.f([], \kappa_0)$$

$$[\![v := e; x]\!](\eta) = \lambda f.f([(v, \mathcal{E}[\![e]\!])], [\![x]\!](\eta))$$

$$[\![\text{ if } b \text{ then } x_1 \text{ else } x_2]\!](\eta) =$$

$$\lambda f.\lambda\sigma.\begin{cases} [\![x_1]\!](\eta)(f)(\sigma) & \text{if } \mathcal{B}[\![b]\!](\sigma) \\ [\![x_2]\!](\eta)(f)(\sigma) & \text{if } \neg\mathcal{B}[\![b]\!](\sigma) \end{cases}$$

$$[\![y]\!](\eta) = \eta(y)$$

$$[\![\text{ letrec } y \text{ be } g \text{ in } x]\!](\eta) = [\![x]\!](\eta \mid y \mapsto fix(\lambda\phi.[\![g]\!](\eta \mid y \mapsto \phi)))$$

$$[\![x_1 \parallel x_2]\!](\eta) = [\![x_1]\!](\eta) \parallel [\![x_2]\!](\eta)$$

Let $f_0 = fix(\Psi)$ ($f_0 \in \mathbf{F}$) be as in Definition 4.1, and $\eta_0 \in \mathbf{Env}$, $\eta_0(y) = \lambda f.f([], \kappa_0), \forall y \in Y$. We define $\mathcal{D}[\![\cdot]\!] : X \to \Sigma \to \mathbf{P}$ as follows:

$$\mathcal{D}[\![x]\!](\sigma) = [\![x]\!](\eta_0)(f_0)(\sigma)$$

In Definition 4.3 a fixed-point construction is used in the equation that gives the semantics of the letrec construction. Well-definedness of $[\![\cdot]\!]$ is established in Lemma 4.4.

Lemma 4.4: The function $\lambda\phi.[\![g]\!](\eta \mid y \mapsto \phi)$ is $\frac{1}{2}$-contractive in ϕ, for any $g \in G$.

Proof Ommited. A similar lemma is proved in [2] (chapter 8). We recall that $g \in G$ is a *guarded* $\mathcal{L}$ statement. □

Examples 4.5: We consider two $\mathcal{L}$ example programs. In the first example, three parallel processes run independently (using different variables). Due to the presence of a shared variable, the second program is non-deterministic. In the both cases the semantics is computed according to the non-interleaving interpretation of the parallel composition operator. For easier readability, instead of $a_1; (a_2; \cdots; (a_n; x))$ we write $a_1; a_2; \cdots; a_n; x$.

(a) Let $x \in X$,

$x =$ letrec y_1 be
  if $(v_1 < 3)$ then $v_1 := v_1 + 1; y_1$
  else skip
in letrec y_2 be
  if $(v_2 < 3)$ then $v_2 := v_2 + 1; y_2$
  else skip
in letrec y_3 be
  if $(v_3 < 3)$ then $v_3 := v_3 + 1; y_3$
  else skip
in $v_1 := 0; v_2 := 0; v_3 := 0; ((y_1 \parallel y_2) \parallel y_3)$

For any $\sigma \in \Sigma$, let
$\sigma_0 = (\sigma \mid v_1 \mapsto 0)$,
$\sigma_1 = (\sigma \mid v_1 \mapsto 0 \mid v_2 \mapsto 0)$,
$\sigma_2 = (\sigma \mid v_1 \mapsto 0 \mid v_2 \mapsto 0 \mid v_3 \mapsto 0)$,
$\sigma_3 = (\sigma \mid v_1 \mapsto 1 \mid v_2 \mapsto 1 \mid v_3 \mapsto 1)$,
$\sigma_4 = (\sigma \mid v_1 \mapsto 2 \mid v_2 \mapsto 2 \mid v_3 \mapsto 2)$,
$\sigma_5 = (\sigma \mid v_1 \mapsto 3 \mid v_2 \mapsto 3 \mid v_3 \mapsto 3)$.
We have:
$\mathcal{D}[\![x]\!](\sigma) = \{\sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_5\}$

(b) Let $x' \in X$
$x' = (v := 100;$ skip $) \parallel (v := 200;$ skip $)$.
For any $\sigma \in \Sigma$, let
$\sigma_1 = (\sigma \mid v \mapsto 100)$ and
$\sigma_2 = (\sigma \mid v \mapsto 200)\}$.
We have:
$\mathcal{D}[\![x']\!](\sigma) = \{\sigma_1\sigma_1, \sigma_2\sigma_2\}$

V. A Haskell implementation of the denotational semantics

We present a semantic interpreter for the language $\mathcal{L}$ which is a direct implementation of the denotational semantics introduced in Section IV. The semantic interpreter is implemented in Haskell [11]. In the sequel we only present the implementation of the denotational mapping and the main semantic operators. The reader can easily provide definitions for the missing functions, which are either very simple or well-known. For example, we do not provide implementations for the set union operation and for the operation that computes the permutations of a multiset. However, the complete Haskell code of the semantic interpreter is available from [22].

A. Syntax and auxiliary operators

The abstract syntax of $\mathcal{L}$ can be implemented in Haskell as follows. We use the types `V`, `Y`, `X`, `A` and `Act` to implement the sets V (of variables) Y (of procedure variables), X (of $\mathcal{L}$ statements), A (of assignment statements) and Act (of partially evaluated assigment statements) respectively.

```
type V    = String
type Y    = String

data A    = A V Exp
data Act  = Act V (S -> Val)

data X    = Skip
          | Prefix A X
          | If Bexp X X
          | Call Y
          | Letrec Y X X
          | Par X X
```

We do not provide here definitions for the types `Exp` and `Bexp`. The type `Exp` implements the class Exp of (numeric) expressions. The type `Bexp` implements the class $Bexp$ of boolean expressions. Also, we assume given two valuations `evalE :: Exp -> S -> Val`, and `evalB :: Bexp -> S -> Bool`. `evalE` and `evalB` implement the mappings $\mathcal{E}[\![\cdot]\!] : Exp \to \Sigma \to \mathbb{Z}$, and $\mathcal{B}[\![\cdot]\!] : Bexp \to \Sigma \to \{ \mathsf{true}, \mathsf{false} \}$, respectively.

```
type Val = Int
type S = [(V,Val)]
```

The type `Val` implements the set $\mathbb{Z}$ of integer values. The type `S` implements the type Σ of states. For testing and evaluation purposes it is convenient to implement the concept of a state as an association list (which associates a value to each variable).

```
subs :: V -> Val -> S -> S
subs v val [] = [(v,val)]
subs v val ((v',val'):s) =
   if (v' == v)
   then (v,val):s
   else (v',val'):subs v val s

assign :: [Act] -> S -> S
assign [] s = s
assign (Act v xi:vvs) s =
   assign vvs (subs v (xi s) s)
```

The function `assign` implements the operation $assign : (A^* \times \Sigma) \to \Sigma$ presented in Section IV.

```
subse :: Eq a =>
         (a -> b) -> a -> b -> (a -> b)
subse f a b a' =
   if (a' == a) then b else f a'
```

The function `subse` implements the notation $(f \mid a \mapsto b)$ introduced in Section II. In this Haskell implementation we only use the mapping `subse` to handle semantic environments.

B. Final semantic domain

The types Q and P implement the domains $\mathbf{Q}$ and $\mathbf{P}$, respectively. We recall that $\mathbf{P}$ is the final semantic domain in our continution-based model. We use the function `prefix` to implement the notation $\sigma \cdot p$. We ommit here the definition of the set union operation `union`. The function `bigunion` implements a finite union.

```
type Q = [S]
type P = [Q]

prefix :: a -> [[a]] -> [[a]]
prefix a ass =
   [ a:as | as <- ass]

bigunion :: Eq a => [[a]] -> [a]
bigunion [xs] = xs
bigunion (xs:xss) =
   xs `union` (bigunion xss)
```

C. Semantic domain and semantic operators in continuation semantics

We use the types `D`, `K`, `F` and `Env` as implementations of the domains $\mathbf{D}$ (of denotations), $\mathbf{K}$ (of asynchronous continuations), $\mathbf{F}$ (of synchronous continuations) and $\mathbf{Env}$ (of semantic environments). We implement multisets (of elementary assignment statements) as Haskell lists. We use the Haskell list concatenation operator `++` to implement the multiset sum operation.

```
type D = F -> S -> P
data K = K D | K0
type F = ([Act],K) -> S -> P
type Env = Y -> D
```

The implementation of the continuation-based semantic operators in Haskell is also straightforward. The functions `park`, `lsyn` and `par` implement the operators $|||$, $\lfloor$ and $\|$, respectively.

```
park :: K -> K -> K
park K0 K0 = K0
park K0 d  = d
park d  K0 = d
park (K d1) (K d2) = K (d1 `par` d2)

lsyn :: D -> D -> D
lsyn d1 d2 = \f ->
   d1 (\(as1,k1) ->
     d2 (\(as2,k2) ->
         f (as1 ++ as2,
              k1 `park` k2)))

par :: D -> D -> D
par d1 d2 =
   \f s -> (lsyn d1 d2 f s)
           `union` (lsyn d1 d2 f s)
```

D. Denotational semantics

The function `upd` implements the mapping upd. The implementation of the mapping `perm` is ommited. `perm` computes all permutations of a multiset represented as a Haskell list.

```
upd :: [Act] -> S -> [S]
upd as s = aux (perm as)
   where
       aux []         = []
       aux (as':ass') =
          [(assign as' s)]
          `union`
          (aux ass')
```

Following the mathematical specification given in Section IV, the initial synchronous continuation `f0` is defined as fixed point of a higher-order mapping `psi`.

```
psi :: F -> F
psi f (as,k) s =
   bigunion [ prefix s'
                (case k of
                      K0    -> [[]]
                      (K d) -> d f s')
            | s' <- upd as s ]
```

```
f0 :: F
f0 = fix psi
```

In a language that supports lazy evaluation, like Haskell, it is possible to implement the fixed-point combinator `fix` according to its defining equation.

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Finally, we implement the denotational semantics $[\![\cdot]\!] : X \to \mathbf{D}$ and the mapping $\mathcal{D}[\![\cdot]\!] : X \to \Sigma \to \mathbf{P}$ as follows:

```
sem :: X -> Env -> D
sem Skip                  e =
   \f -> f ([],K0)
sem (Prefix (A v exp) x) e =
   \f -> f ([Act v (evalE exp)],
            K (sem x e))
sem (If b x1 x2)          e =
   \f -> \s ->
     if (evalB b s)
     then sem x1 e f s
     else sem x2 e f s
sem (Call y)              e = e y
sem (Letrec y x1 x2)      e =
   sem x2
     (subse e y
         (fix (\d ->
               sem x1 (subse e y d))))
sem (Par x1 x2)           e =
   (sem x1 e) `par` (sem x2 e)

den :: X -> S -> P
den x s =
   let f0 = fix psi
       e0 y = \f -> f ([],K0)
   in  sem x e0 f0 s
```

E. Testing the semantic interpreter

Instead of using mathematical notation, in this section we use Haskell as a metalanguage for denotational semantics. Our denotational semantics is implemented in the form of an executable intepreter that can be easily tested and evaluated.

In the following experiments we consider that `s0 = []` is the initial state (`s0 :: S`). The Haskell implementation of the $\mathcal{L}$ example programs $x, x' (\in X)$ given in Example 4.5(a) is available from [22]. Let `x :: X` be the Haskell implementation of $x \in X$. Let `x' :: X` be the Haskell implementation of $x' \in X$. Running `x` with `den` produces the following output:

```
den x s0  ⇒
[[[("v1",0)],[("v1",0),("v2",0)],
   [("v1",0),("v2",0),("v3",0)],
   [("v1",1),("v2",1),("v3",1)],
   [("v1",2),("v2",2),("v3",2)],
   [("v1",3),("v2",3),("v3",3)],
   [("v1",3),("v2",3),("v3",3)]]]
```

Also, running `x'` with `den` produces the following output:

```
den x' s0  ⇒
 [[[("v1",200)],[("v1",200)]],
  [[("v1",100)],[("v1",100)]]]
```

VI. Conclusion

It is well-known that traditional continuations [14] ca be used to model a variety of advanced control concepts, including non-local exits, coroutines and even multitasking [21]. However, the traditional continuations do not work well enough in the presence of concurrency [8]. In [15] we introduced a continuation semantics for concurrency (CSC); in subsequent work we refined the CSC technique [6], [16], [20]. In this paper we showed that the technique of continuations can be used to express maximal parallelism in a compositional manner. We presented a denotational semantics designed with metric spaces and continuations for a concurrent imperative language in which the semantic operator for parallel composition is designed according to the maximal parallelism model of non-interleaved computations. The denotational semantics was developed by using the continuation-based technique introduced in [20]. We also presented a Haskell implementation of our denotational semantics.

References

[1] P. America, J.J.M.M. Rutten, Solving reflexive domain equations in a category of complete metric spaces, *Journal of Computer and System Sciences*, vol. 39(3), pp. 343–375, 1989.

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

[3] J.W. De Bakker, J.H.A. Warmerdam, Metric pomset semantics for a concurrent language with recursion, *LNCS*, vol. 469, pp. 21–49, Springer, 1990.

[4] G. Ciobanu, E.N. Todoran, Metric denotational semantics for parallel rewriting of multisets, *Proceedings of 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2011)*, pp. 276–283, IEEE Computer Press, 2011.

[5] G. Ciobanu, E.N. Todoran, Relating two metric semantics for parallel rewriting of multisets, *Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*, pp. 273–280, IEEE Computer Press, 2012.

[6] G. Ciobanu, E.N. Todoran, Continuation semantics for asynchronous concurrency, *Fundamenta Informaticae*, vol. 131(3-4), pp. 373-388, 2014.

[7] G. Ciobanu, E.N. Todoran, Continuation semantics for concurrency with multiple channels communication, Proceedings of 17th International Conference on Formal Engineering Methods (ICFEM 2015), *Lecture Notes in Computer Science*, vol. 9407, pp. 400–416, Springer, 2015.

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

[9] R. Milner, *Communication and concurrency,* Prentice Hall, 1989.

[10] R. Milner, *Communicating and mobile systems: the π-calculus,* Cambridge University Press, 1999.

[11] S. Peyton Jones, J. Hughes, editors, Report on the Programming Language Haskell 98: A Non-Strict Purely Functional Language, 1999; Available from `http://www.haskell.org/`.

[12] G. Plotkin, Call-by-name, call-by-value and the λ-calculus, *Theoretical Computer Science*, vol. 1, pp. 125–159, 1975.

[13] J. Reynolds, Definitional interpreters for higher-order programming languages, *25th ACM National Conference*, pp. 717–740, 1972.

[14] C. Stratchey, C. Wadsworth, Continuations: a mathematical semantics for handling full jumps, *Journal of Higher-Order and Symbolic Computation*, vol. 13(1), pp. 135–152, 2000.

[15] E.N. Todoran, Metric semantics for synchronous and asynchronous communication: a continuation-based approach, *Electronic Notes in Theoretical Computer Science*, vol. 28, pp. 101–127, Elsevier, 2000.

[16] E.N.Todoran, N. Papaspyrou, Continuations for parallel logic programming, *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pp. 257-267, 2000.

[17] E.N. Todoran, N. Papaspyrou, Continuations for prototyping concurrent languages, Technical Report CSD-SW-TR-1-06, National Technical University of Athens, Software Engineering Laboratory, 2006.

[18] E.N. Todoran, Comparative semantics for modern communication abstractions, *Proceedings of 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing (ICCP 2008)*, pp. 153–160, 2008.

[19] E.N. Todoran, C. Adam, M. Balc, R. Pop, R. Radu, D. Simina, E. Varga, D.A. Zaharia, Mobile objects and modern communication abstractions: design issues and denotational semantics, *Proceedings of 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, pp. 191–198, IEEE Computer Press, 2011.

[20] E.N. Todoran, N. Papaspyrou, Experiments with continuation semantics for DNA computing, *Proceedings of 2013 IEEE 9th International Conference on Intelligent Computer Communication and Processing (ICCP 2013)*, pp. 251–258, 2013.

[21] M. Wand, Continuation-based multiprocessing, *Higher-Order and Symbolic Com- putation*, vol. 12(3), 285-299, 1999.

[22] `ftp://ftp.utcluj.ro/pub/users/gc/acam2015`