Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Comparative Semantics for Linear Arrays of Communicating Processes

J.W. de Bakker, F. van Breugel, A. de Bruin

Computer Science/Department of Software Technology

CS-R9336 1993

Comparative Semantics for Linear Arrays of Communicating Processes
a study of the UNIX* fork and pipe commands

J.W. de Bakker[1,2], F. van Breugel[1,2] and A. de Bruin[3]

[1] *CWI*

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

[2] *Vrije Universiteit*

P.O. Box 7161, 1007 MC Amsterdam, The Netherlands

[3] *Erasmus Universiteit*

P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Abstract

Operational ($\mathcal{O}$) and denotational ($\mathcal{D}$) semantic models are designed for a language incorporating a version of the UNIX fork and pipe commands. Taking a simple while language as starting point, a number of programming constructs are added which achieve that a program can generate a dynamically evolving linear array of processes connected by channels. Over these channels sequences of values ('streams') are transmitted. Both $\mathcal{O}$ and $\mathcal{D}$ are defined as (unique) fixed point of a contractive higher order operator. This allows a smooth proof that $\mathcal{O}$ and $\mathcal{D}$ are equivalent. Additional features are the use of hiatons, and of the closely related syntactic resumptions and semantic continuations.

AMS Subject Classification (1991): 68Q55
CR Subject Classification (1991): D.3.1, F.3.2
Keywords & Phrases: operational semantics, denotational semantics, complete metric space, fork statement, hiaton, resumption, continuation
Note: The work of F. van Breugel was partially supported by the Netherlands *Nationale Faciliteit Informatica* programme, project Research and Education in Concurrent Systems (REX). This paper will appear in Proceedings of the Eighteenth International Symposium on Mathematical Foundations of Computer Science, Gdansk, August 30 - September 3, 1993.

Introduction

We present a comparative semantic study of a simple imperative language $\mathcal{L}$ which features the construction of dynamically evolving linear arrays of communicating processes. Our investigation was in particular motivated by the UNIX fork and pipe commands which return in somewhat adapted form in $\mathcal{L}$.

Both operational ($\mathcal{O}$) - based on an SOS style transition system ([Plo81]) - and denotational ($\mathcal{D}$) semantics for $\mathcal{L}$ will be presented, and their equivalence will be established. Simple topological techniques will suffice for the mathematical underpinning of both models. In fact, Banach's fixed point theorem ([Ban22]) is all we need. ([BR92] gives an overview of more advanced uses of topological modelling.)

Forks and pipes occur in several papers on programming language design and application (forks, e.g., in [HSS91], pipes in [KK92]). *Semantic* studies focusing on these topics are scarce (e.g. [AW85, Ben82, Bru86, MA89, RS83, RS92]), and none of them develops *both* operational and denotational models. Accordingly, we see the *comparative* result as the main contribution of our paper.

*UNIX is a trademark of Bell Laboratories.

In the remainder of this introduction we informally introduce $\mathcal{L}$, and present three simple examples of its use culminating in a version of the sieve of Eratosthenes. Sections 1 and 2 present the operational and denotational semantics, respectively. In the design of $\mathcal{O}$, arrays of processes are modelled using the concept of (nested) *resumptions.* For $\mathcal{D}$, *continuations* are an essential tool. In Section 3, we prove the equivalence of $\mathcal{O}$ and $\mathcal{D}$ using the unique fixed point proof principle from [KR90]. Let us mention one subtlety in the semantic models: in order to apply Banach's theorem, we require contractiveness at various instances. At appropriate points a version of Park's *hiaton* ([Par83]) is used to enforce contractiveness if this would not arise naturally.

We now present the syntax of $\mathcal{L}$. It is a simple imperative language with assignment, while statements and the like, to which three further constructs are added: $\mathbf{write}\,(e)$, $\mathbf{read}\,(v)$, and $\mathbf{fork}\,(v)$. The syntax for $\mathcal{L}$ follows

$$s ::= v := e \mid \mathbf{skip} \mid \mathbf{write}\,(e) \mid \mathbf{read}\,(v) \mid \mathbf{fork}\,(v) \mid s\,;s \mid \mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}.$$

In the sequel, a program in execution will be called a *process.* Each process has exactly one input channel and one output channel connected to it (see Figure 1). Execution of the *write statement* $\mathbf{write}\,(e)$ has the effect that the value of the expression e is written on the output channel, the effect of the *read statement* $\mathbf{read}\,(v)$ is that a new value is read from the input channel which is then assigned to the variable v. If there are no more values on the input channel then the process blocks (terminates).

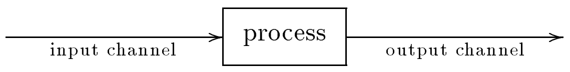

Figure 1

A process can be modelled by a function which takes an input stream as an argument and yields an output stream as a result. The input stream is the sequence of all values assumed to be preloaded on the input channel, and the output stream is the sequence of all values to be written by the process on the output channel. Both streams can very well be infinite, and this means that nonterminating processes are meaningful in this setting. We give as first example, a '2-filter' described by the program

```
while true
do read (v);
    if v mod 2 ≠ 0 then write (v) else skip fi
od.
```

This program filters all even numbers, passing only the odd numbers from its input channel to its output channel.

The other new concept in the language is the *fork statement*, described by a statement of the form $\mathbf{fork}\,(v)$. This statement can be regarded as a combination of the UNIX fork and the UNIX pipe. When a process executes the statement $\mathbf{fork}\,(v)$, the effect is that an almost identical copy of the process is constructed. We call the original process the *parent* and the new process the *child.* After the fork statement has been evaluated both processes continue execution with the statement following the fork statement. There is no sharing of variables, each process has its own set of variables all (but for the variable v, see below) having the values they had in the parent process when the fork statement was executed.

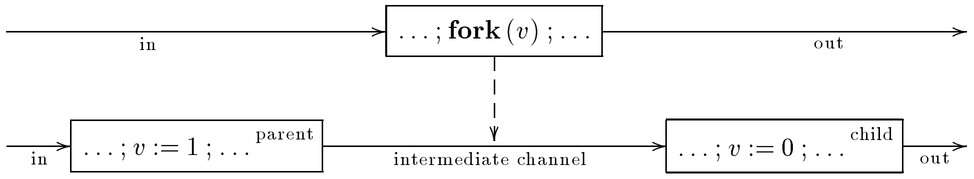

Figure 2

There are two differences between the two processes. The first one has to do with the fact that executing **fork** (v) has as a side effect that a value is assigned to v. In the parent process the assignment $v := 1$ is performed, in the child process the value 0 is assigned to v. The other difference has to do with the input and output channels of the original process. On execution of the fork statement a new intermediate channel is constructed which behaves like a UNIX pipe. The parent process remains connected to the original input channel, but from now on writes on the new intermediate channel. The child will write on the original output channel, but reads from the intermediate channel. The effect of a fork statement is depicted in Figure 2. The second example is the program

```
read (v);
write (v);
fork (w);
if w = 1
then while true
        do read (v);
            if v mod 2 ≠ 0 then write (v) else skip fi
        od
else  while true
        do read (v);
            if v mod 3 ≠ 0 then write (v) else skip fi
        od
fi.
```

The original process passes one value from the input to the output unaltered, and then splits into two filters: the parent filters out all even numbers, passing only the odd input numbers to the child. The child filters out all the numbers which are a multiple of 3. The effect is a filter that passes its first input number unaltered, and then passes only those inputs values that are not multiples of 2 or 3.

The final example is a version of the sieve of Eratosthenes:

```
read (v);
while true
do read (v);
    write (v);
    fork (w);
    if w = 1
    then while true
            do read (x);
                if x mod v ≠ 0 then write (x) else skip fi
            od
    else  skip
    fi
od
```

If on the input channel for the original process the stream of the positive natural numbers is inserted, then execution of this program will result in an expanding array of processes which in cooperation yield an output stream consisting of all prime numbers. The original process can be called an 'expander' (e in Figure 3), it reads a number n and expands into a filter process (the parent) which blocks all multiples of n (the parent process is denoted by n in Figure 3), and a new expander process (the child) which behaves like the original process. How this array evolves is shown in Figure 3.

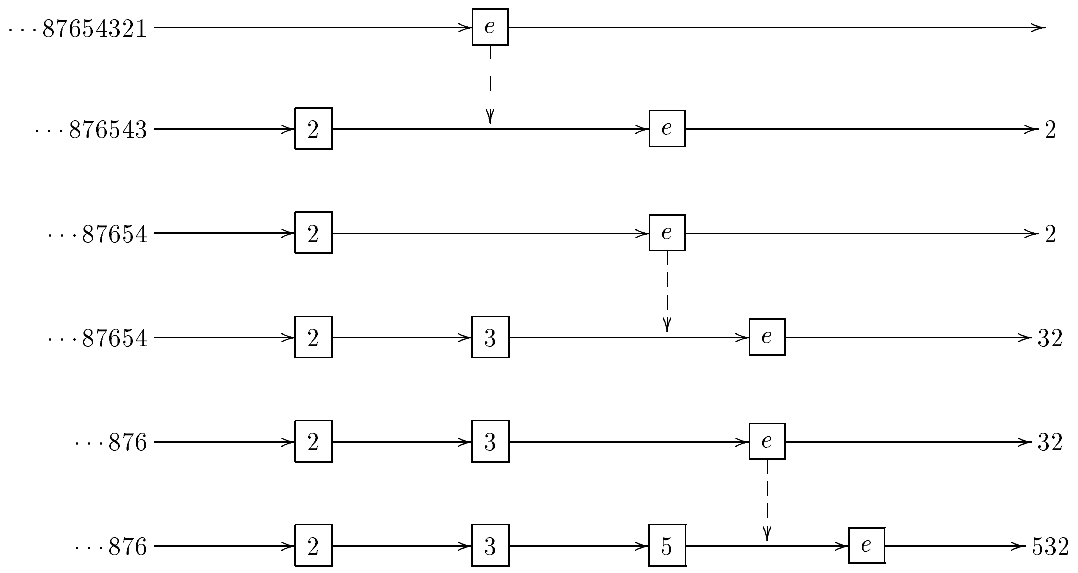

Figure 3

1. Operational Semantics

Before we come to the operational semantics, we first repeat the definition of the syntax for $\mathcal{L}$. Let $(v \in)$ *Var* be the syntactic class of *variables*.

Definition 1.1 The language $(s \in) \mathcal{L}$ is defined by

$$s ::= v := e \mid \textbf{skip} \mid \textbf{write}\,(e) \mid \textbf{read}\,(v) \mid \textbf{fork}\,(v) \mid s\,;s \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi} \mid \textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}.$$

Here e and b range over the syntactic classes of *expressions* and *boolean expressions*, respectively. We assume a simple syntax for these which we do not bother to specify. Programs in $\mathcal{L}$ operate on streams of input values, delivering streams of output values. Let us use $(\alpha \in)$ *Val* to denote the set of these (input and output) *values*. In addition, we shall have occasion to use the 'silent' value τ. We write $(\beta \in)\ Val_\tau = Val \cup \{\tau\}$. The role of the τ-value - sometimes also called *hiaton* - will be, in the transition system to be introduced in a moment, to signal a 'silent' transition. Such a transition does not correspond to delivering a 'normal' value (from *Val*); it is employed in a situation where the metric framework requires a step to achieve contractiveness.

The operational semantics for $\mathcal{L}$ will be based on a transition system in the familiar SOS style. In this system we encounter

- The set $(\sigma \in)\ State = Var \rightarrow Val$ of *states*. The notation $\sigma\{\alpha/v\}$ is used for a state which is like σ, but for its value in v which equals α.

- The set $(\varsigma \in)\ \mathit{Val}_\tau^\infty$, consisting of all finite and infinite sequences (the 'streams' mentioned earlier) of elements from Val_τ.
- The special symbol E standing for *termination*.
- Auxiliary syntactic categories of so-called *resumptions* and *nested resumptions*. These are introduced in

DEFINITION 1.2 The class of resumptions $(r \in)\ \mathit{Res}$ is defined by

$$r ::= \mathrm{E} \mid s : r.$$

The class of nested resumptions $(\rho \in)\ \mathit{NRes}$ is defined by

$$\rho ::= \varsigma \mid < r,\, \sigma,\, \rho > .$$

Resumptions are sequences of statements ending in E. Nested resumptions have a structure of the form

$$\rho =< r_1,\, \sigma_1,\, < r_2,\, \sigma_2,\, \ldots < r_n,\, \sigma_n,\, \varsigma > \ldots >> .$$

Nested resumptions correspond to process arrays as described in the introduction in the following way:

- For $n = 0$, we have that $\rho = \varsigma$. In this case ρ consists of no more than the input stream ς.
- If $n = 1$, then $\rho =< r_1,\, \sigma_1,\, \varsigma >$. The process ρ executes the (sequence of) statements specified by r_1, for state σ_1 and input stream ς.
- For $n = 2$, we obtain $\rho =< r_1,\, \sigma_1,\, < r_2,\, \sigma_2,\, \varsigma >>$. In this case, ρ consists of a parent process $\rho_p =< r_2,\, \sigma_2,\, \varsigma >$ - interpreted as just described - the output of which acts as input for the child process $(\rho =)\, \rho_c =< r_1,\, \sigma_1,\, \rho_p >$.
- For $n > 2$, we obtain a process array of length n as described above.

In the transition system $\mathcal{T}$ to be presented in the next definition, we use $\mathcal{V}(e)(\sigma)$ (yielding an element in Val) and $\mathcal{B}(b)(\sigma)$ to denote the values of e and b in state σ.

DEFINITION 1.3 The transition system $\mathcal{T} = (\mathit{NRes},\, \mathit{Val}_\tau, \rightarrow)$ has NRes as the set of its configurations and Val_τ as its set of labels. The transition relation $\rightarrow$ is the smallest subset of $\mathit{NRes} \times \mathit{Val}_\tau \times \mathit{NRes}$ satisfying the rules given below. We use the notation

$$\rho \xrightarrow{\beta} \rho'$$

as short hand for $(\rho, \beta, \rho') \in \rightarrow$. A rule of the form

$$\text{if } \rho_1 \xrightarrow{\beta} \rho \text{ then } \rho_2 \xrightarrow{\beta} \rho$$

will be abbreviated to $\rho_2 \rightarrow_0 \rho_1$; the 0-subscript indicates that we have here a *zero-step* transition.

(1) $< (v := e) : r,\, \sigma,\, \rho > \rightarrow_0 < r,\, \sigma\{\alpha/v\},\, \rho >$, where $\alpha = \mathcal{V}(e)(\sigma)$

(2) $< \textbf{skip} : r,\, \sigma,\, \rho > \rightarrow_0 < r,\, \sigma,\, \rho >$

(3) $< \textbf{write}(e) : r,\, \sigma,\, \rho > \xrightarrow{\alpha} < r,\, \sigma,\, \rho >$, where $\alpha = \mathcal{V}(e)(\sigma)$

(4) if $\rho \xrightarrow{\alpha} \rho'$ then $< \textbf{read}(v) : r,\, \sigma,\, \rho > \xrightarrow{\tau} < r,\, \sigma\{\alpha/v\},\, \rho' >$

(5) if $\rho \xrightarrow{\tau} \rho'$ then $< \textbf{read}(v) : r,\, \sigma,\, \rho > \xrightarrow{\tau} < \textbf{read}(v) : r,\, \sigma,\, \rho' >$

(6) $< \textbf{fork}(v) : r,\, \sigma,\, \rho > \rightarrow_0 < r,\, \sigma\{0/v\},\, < r,\, \sigma\{1/v\},\, \rho >>$

(7) $< (s_1\, ;\, s_2) : r,\, \sigma,\, \rho > \rightarrow_0 < s_1 : (s_2 : r),\, \sigma,\, \rho >$

(8) if $\mathcal{B}(b)(\sigma)$ then $< \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r,\, \sigma,\, \rho > \rightarrow_0 < s_1 : r,\, \sigma,\, \rho >$

(9) if $\neg\, \mathcal{B}(b)(\sigma)$ then $< \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r,\, \sigma,\, \rho > \rightarrow_0 < s_2 : r,\, \sigma,\, \rho >$

(10) $< \textbf{while } b \textbf{ do } s \textbf{ od} : r,\, \sigma,\, \rho > \xrightarrow{\tau} < \textbf{if } b \textbf{ then } s\, ;\, \textbf{while } b \textbf{ do } s \textbf{ od else skip fi} : r,\, \sigma,\, \rho >$

(11) $\beta \cdot \varsigma \xrightarrow{\beta} \varsigma$

We add some explanations:

- A transition $\rho \xrightarrow{\beta} \rho'$ expresses that (the process corresponding to) ρ performs a one-step transition to process ρ', while producing a value β (either a normal or a silent value) which is appended to the current output stream.

- Note that there is no transition defined for a configuration $< \mathrm{E},\, \sigma,\, \rho >$. As a consequence, neither is there a transition possible for, e.g., $< (v := e) : \mathrm{E},\, \sigma,\, \rho >$, $< \textbf{read}(v) : \mathrm{E},\, \sigma,\, < \mathrm{E},\, \sigma,\, \rho >>$, etc. We emphasize that transitions become observable only by delivering output values (including an occasional silent value); note that this is quite different from more customary models where state changes - from σ to some σ' - are observable.

- The rules for $v := e$, **skip**, $s_1\, ;\, s_2$, and **if** b **then** s_1 **else** s_2 **fi** should be clear. The while statement always induces a silent step. (A zero-step transition would not work in this case, this being incompatible with a subsequent crucial property of zero-step transitions, cf. Lemma 1.7.)

- The effect of **write**$(e) : r$ is to append $\alpha\, (= \mathcal{V}(e)(\sigma))$ to the output stream, and continue with r.

- For a **read**(v) statement - with respect to current r, σ, and ρ - we distinguish two cases. In the 'normal' situation, an input α is available, produced (as output) by ρ when it turns itself into ρ'. We then assign α to v, and continue with r, the updated state $\sigma\{\alpha/v\}$, and the new parent process ρ'. Otherwise, i.e. when ρ produces a silent step τ, we reject this as possible value for v - recall that the codomain of any state equals Val rather than Val_τ -, maintain the requirement for an input **read**(v), and continue with r, σ, and parent process ρ'. (As for the while statement also in this case a zero-step transition would not work.)

- The fork statement **fork**(v) - with respect to current r, σ, and ρ - creates two processes, the parent process

  $$\rho_p = < r,\, \sigma\{1/v\},\, \rho >$$

  and the child process

  $$\rho_c = < r,\, \sigma\{0/v\},\, \rho_p > .$$

We observe that

- ∗ The forking process performs a zero-step transition to ρ_c.
- ∗ Both ρ_p and ρ_c execute the resumption r.
- ∗ In ρ_p, the fork variable is set to 1, in ρ_c it is set to 0. This offers the possibility to 'program' in r so as to have different executions in ρ_p and ρ_c, respectively (cf. the examples in the introduction).
- ∗ Since ρ_p occurs as part of ρ_c, the net effect of this is that the output of ρ_p acts as input for ρ_c, cf. also the way the read and write rules are defined.

- The final rule simply describes how an input stream $\beta \cdot \varsigma$ performs a one step transition delivering the output β, and turns itself into ς.
- The transition system $\mathcal{T}$ specifies deterministic behaviour (see Lemma 1.8) and synchronous communication. Concerning the former phenomenon, adding the metarule

  $$\text{if } \rho \rightarrow_0 \rho' \text{ then } < r, \sigma, \rho > \rightarrow_0 < r, \sigma, \rho' >$$

  would allow some form of parallelism in the execution of processes. As a consequence of the latter phenomenon, a parent process can only write when its child is willing to read. As we will see, a communication between a parent and its child will not be visible in the operational semantics (apart from a silent transition). Asynchronous communication could be handled by adding an output sequence to the nested resumptions which then take the form $< \varsigma, r, \sigma, \rho >$. A study of these variations is outside the scope of the present paper.

We now describe how to obtain the operational semantics $\mathcal{O} : \mathcal{L} \rightarrow \mathit{Proc}$, where $\mathit{Proc} = \mathit{State} \rightarrow \mathit{Val}_\tau^\infty \rightarrow \mathit{Val}_\tau^\infty$. We see that $\mathcal{O}[\![s]\!](\sigma)$ yields a function transforming streams to streams, in accordance with the intended model for $\mathcal{L}$. We shall employ an intermediate mapping $\mathcal{O} : \mathit{NRes} \rightarrow \mathit{Val}_\tau^\infty$; $\mathcal{O}$ is the function which, for argument ρ, collects the sequence of labels produced successively by the transitions as specified by $\mathcal{T}$, starting from ρ. Thus $\mathcal{O}(\rho) = \varsigma$ states that the process ρ yields output stream ς. (Recall that the *input* to ρ is included in its own description.) Let us use the terminology ρ *blocks* in case ρ cannot make any transitions, that is

$$\neg \exists\, \beta, \rho' : \rho \xrightarrow{\beta} \rho'.$$

As defining properties for $\mathcal{O}$ we want the following to be satisfied:

$$\mathcal{O}(\rho) = \begin{cases} \varepsilon & \text{if } \rho \text{ blocks} \\ \beta \cdot \mathcal{O}(\rho') & \text{if } \rho \xrightarrow{\beta} \rho' \end{cases}$$

Note that ρ' is not necessarily of smaller syntactic complexity than ρ, so this 'definition' cannot be shown to be well-formed simply by structural induction on ρ. Instead, we use a familiar technique for dealing with recursive definitions, viz. through the use of fixed points of some higher-order operator. Let Φ be an operator which maps meanings ϕ to meanings ϕ' in the following way:

Definition 1.4 Let $(\phi \in)\, \mathit{Sem}_{\mathcal{O}} = \mathit{NRes} \rightarrow \mathit{Val}_\tau^\infty$, and let $\Phi : \mathit{Sem}_{\mathcal{O}} \rightarrow \mathit{Sem}_{\mathcal{O}}$ be defined as follows:

$$\Phi(\phi)(\rho) = \begin{cases} \varepsilon & \text{if } \rho \text{ blocks} \\ \beta \cdot \phi(\rho') & \text{if } \rho \xrightarrow{\beta} \rho' \end{cases}$$

Well-definedness of this definition requires that $\mathcal{T}$ is deterministic, i.e. that each ρ can make at most one transition. Lemma 1.8 below states this result.

By the definition of Φ, it is immediate that it is contractive[1] in ϕ. Since $Sem_{\mathcal{O}}$ is a complete metric space[2], we have, by Banach's theorem[3], that Φ has a unique fixed point, and we have justified

DEFINITION 1.5 The operational semantics $\mathcal{O} : Sem_{\mathcal{O}}$ is defined by

$$\mathcal{O} = \mathit{fix}\,(\Phi).$$

In addition to its serving as a means to define $\mathcal{O}$, Φ will play a crucial role (in Section 3) in the proof that $(*)$ $\mathcal{O} = \mathcal{D}$ (the denotational semantics to be introduced in Section 2). In fact, $(*)$ follows as an immediate corollary of an argument exploiting the unique fixed point property of Φ.

The next step in the technical development is the introduction of the complexity measure $c : NRes \to \mathbb{N}$ in

DEFINITION 1.6 The complexity measure $c : NRes \to \mathbb{N}$ is defined by

$$c(\varsigma) = 1 \qquad\qquad c(< r, \sigma, \rho >) = c(r) + c(\rho)$$

where

$$c(\mathrm{E}) = 1 \qquad\qquad c(s : r) = c(s) * c(r)$$

where

$$\begin{array}{lll} c(v := e) &= 2 & c(\mathbf{fork}\,(v)) &= 3 \\ c(\mathbf{skip}) &= 2 & c(s_1\,;\,s_2) &= c(s_1) * c(s_2) + 1 \\ c(\mathbf{write}\,(e)) &= 1 & c(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}) &= c(s_1) + c(s_2) \\ c(\mathbf{read}\,(v)) &= 1 & c(\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}) &= 2 \end{array}$$

The measure c is used in the proof of the following two lemmas.

LEMMA 1.7 *For all ρ and ρ', if $\rho \to_0 \rho'$ then $c(\rho) > c(\rho')$.*

[1] Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \to X'$ is called contractive if there exists an δ, with $0 \le \delta < 1$, such that, for all x and x',

$$d_{X'}(f(x), f(x')) \le \delta \cdot d_X(x, x').$$

[2] The set Val_τ^∞ is endowed with the metric

$$d(\varsigma, \varsigma') = \begin{cases} 0 & \text{if } \varsigma = \varsigma' \\ 2^{-n} & \text{otherwise} \end{cases}$$

where n is the longest common prefix of the sequences ς and ς'. By means of this metric we can endow $Sem_{\mathcal{O}}$ with the metric

$$d(\phi, \phi') = \sup\{d(\phi(\rho), \phi'(\rho)) \mid \rho \in NRes\}.$$

These metrics are ultrametrics, i.e., for all x, x', and x'',

$$d(x, x'') \le \max\{d(x, x'), d(x', x'')\}.$$

[3] Let (X, d_X) be a complete metric space. If $f : X \to X$ is contractive then f has a unique fixed point $\mathit{fix}\,(f)$.

Proof Only a few cases of the proof of this lemma are elaborated on.

1. Let $\rho \equiv < (v := e) : r,\, \sigma,\, \bar{\rho} >$. Then
$$\begin{array}{ll} c\,(< (v := e) : r,\, \sigma,\, \bar{\rho} >) & \\ \quad = & 2 * c\,(r) + c\,(\bar{\rho}) \\ \quad > & c\,(r) + c\,(\bar{\rho}) \\ \quad = & c\,(< r,\, \sigma\{\alpha/v\},\, \bar{\rho} >). \end{array}$$

2. Let $\rho \equiv < \textbf{fork}\,(v) : r,\, \sigma,\, \bar{\rho} >$. Then
$$\begin{array}{ll} c\,(< \textbf{fork}\,(v) : r,\, \sigma,\, \bar{\rho} >) & \\ \quad = & 3 * c\,(r) + c\,(\bar{\rho}) \\ \quad > & 2 * c\,(r) + c\,(\bar{\rho}) \\ \quad = & c\,(< r,\, \sigma\{0/v\},\, < r,\, \sigma\{1/v\},\, \bar{\rho} >>). \end{array}$$

□

Lemma 1.8 *The transition system $\mathcal{T}$ is deterministic.*

Proof We can show that, for all ρ, $|\{\, (\beta, \rho') \mid \rho \xrightarrow{\beta} \rho' \,\}| \leq 1$ by induction on the complexity of ρ. □

We are now ready for the key definition of this section.

Definition 1.9 The operational semantics $\mathcal{O} : \mathcal{L} \to \mathit{Proc}$ is defined by

$$\mathcal{O}\,[\![s]\!] = \lambda\sigma \,.\, \lambda\varsigma \,.\, \mathcal{O}\,(< s : \mathrm{E},\, \sigma,\, \varsigma >).$$

The final program of the introduction with an arbitrary initial state and the input stream 12345678 will produce the output stream $\tau^3 2 \tau^4 3 \tau^8 5 \tau^{10} 7 \tau^7$ and terminate as the reader may verify.

2. Denotational Semantics

The denotational semantics for $\mathcal{L}$ uses the set of *continuations* $(\theta \in) \mathit{Cont} = \mathit{State} \to \mathit{Val}_\tau^\infty \to^1 \mathit{Val}_\tau^\infty$. Note that, but for the specialization to the nonexpansive[4] function space $\to^1$, *Cont* equals *Proc* as introduced earlier. Continuations correspond to resumptions in the sense that, as we shall see in Definition 3.2, meanings of *Res* reside in *Cont*.

We shall use $\mathit{first}\,(\varsigma)$ to denote the first element of the nonempty sequence ς, and $\mathit{rest}\,(\varsigma)$ to denote the result of omitting the first element from the nonempty sequence ς.

The denotational semantics $\mathcal{D}$ for $\mathcal{L}$ is presented in

Definition 2.1 Let $(\psi \in)\, \mathit{Sem}_{\mathcal{D}} = \mathcal{L} \to \mathit{Cont} \to^1 \mathit{Cont}$. Let $\Psi : \mathit{Sem}_{\mathcal{D}} \to \mathit{Sem}_{\mathcal{D}}$ be defined by

[4]Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \to X'$ is called nonexpansive if, for all x and x',
$$d_{X'}\,(f\,(x), f\,(x')) \leq d_X\,(x, x').$$

$$
\begin{array}{lll}
\Psi(\psi)(v := e)(\theta)(\sigma)(\varsigma) & = \theta(\sigma\{\alpha/v\})(\varsigma) & \text{where } \alpha = \mathcal{V}(e)(\sigma) \\
\Psi(\psi)(\mathbf{skip})(\theta)(\sigma)(\varsigma) & = \theta(\sigma)(\varsigma) & \\
\Psi(\psi)(\mathbf{write}(e))(\theta)(\sigma)(\varsigma) & = \alpha \cdot \theta(\sigma)(\varsigma) & \text{where } \alpha = \mathcal{V}(e)(\sigma) \\
\Psi(\psi)(\mathbf{read}(v))(\theta)(\sigma)(\varsigma) & = \begin{cases} \varepsilon & \text{(a)} \\ \tau \cdot \theta(\sigma\{\mathit{first}(\varsigma)/v\})(\mathit{rest}(\varsigma)) & \text{(b)} \\ \tau \cdot \psi(\mathbf{read}(v))(\theta)(\sigma)(\mathit{rest}(\varsigma)) & \text{(c)} \end{cases} & \\
\Psi(\psi)(\mathbf{fork}(v))(\theta)(\sigma)(\varsigma) & = (\theta(\sigma\{0/v\}))(\theta(\sigma\{1/v\})(\varsigma)) & \\
\Psi(\psi)(s_1 ; s_2)(\theta)(\sigma)(\varsigma) & = \Psi(\psi)(s_1)(\Psi(\psi)(s_2)(\theta))(\sigma)(\varsigma) & \\
\Psi(\psi)(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi})(\theta)(\sigma)(\varsigma) & = \begin{cases} \Psi(\psi)(s_1)(\theta)(\sigma)(\varsigma) & \text{(d)} \\ \Psi(\psi)(s_2)(\theta)(\sigma)(\varsigma) & \text{(e)} \end{cases} & \\
\Psi(\psi)(\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od})(\theta)(\sigma)(\varsigma) & = \tau \cdot \psi(\mathbf{if}\ b\ \mathbf{then}\ s ; \mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}\ \mathbf{else}\ \mathbf{skip}\ \mathbf{fi})(\theta)(\sigma)(\varsigma) &
\end{array}
$$

where

(a) if $\varsigma = \varepsilon$
(b) if $\varsigma \neq \varepsilon$ and $\mathit{first}(\varsigma) \neq \tau$
(c) if $\varsigma \neq \varepsilon$ and $\mathit{first}(\varsigma) = \tau$
(d) if $\mathcal{B}(b)(\sigma)$
(e) if $\neg\, \mathcal{B}(b)(\sigma)$

The denotational semantics $\mathcal{D} : \mathit{Sem}_{\mathcal{D}}$ is defined by

$$\mathcal{D} = \mathit{fix}(\Psi).$$

Some remarks:

- Much of the structure of the above clauses may be understood by consulting $\mathcal{T}$. For example, the clause for the fork statement amounts to

$$\mathcal{D}(\mathbf{fork}(v))(\theta)(\sigma)(\varsigma) = (\theta(\sigma\{0/v\}))(\theta(\sigma\{1/v\})(\varsigma)).$$

  Now using the correspondence between the semantic continuation θ and the syntactic resumption r, we see that this is an immediate counterpart of the transition

$$< \mathbf{fork}(v) : r,\ \sigma,\ \varsigma > \rightarrow_0 < r,\ \sigma\{0/v\},\ < r,\ \sigma\{1/v\},\ \varsigma >> .$$

- Similar to what we did for $\mathcal{O}$, we have defined $\mathcal{D}$ here as (unique) fixed point of a higher-order mapping. Such a 'global' fixed point approach is attractive, were it only for symmetry reasons. However, a more traditional ('local') approach, where the taking of fixed points is restricted to the clauses for the read and while statement, would also serve our purposes.

Definition 2.1 is justified in

Lemma 2.2 *For all ψ, s, θ, and σ,*

the mapping $\Psi(\psi)(s)(\theta)(\sigma)$ is nonexpansive (in ς),
the mapping $\Psi(\psi)(s)$ is nonexpansive (in θ), and
the mapping Ψ is contractive (in ψ).

Proof We only consider the second property. It can be shown that, for all ψ, s, θ_1, θ_2, σ, and ς,

$$d(\Psi(\psi)(s)(\theta_1)(\sigma)(\varsigma), \Psi(\psi)(s)(\theta_2)(\sigma)(\varsigma)) \leq d(\theta_1, \theta_2)$$

by structural induction on s. Only a few cases are elaborated on.

1. Let $s \equiv \mathbf{read}(v)$. We distinguish three cases.
   (a) If $\varsigma = \varepsilon$, then
   $$\begin{aligned} & d(\Psi(\psi)(\mathbf{read}(v))(\theta_1)(\sigma)(\varsigma), \Psi(\psi)(\mathbf{read}(v))(\theta_2)(\sigma)(\varsigma)) \\ = {} & d(\varepsilon, \varepsilon) \\ \leq {} & d(\theta_1, \theta_2). \end{aligned}$$
   (b) If $\varsigma \neq \varepsilon$ and $\mathit{first}(\varsigma) \neq \tau$, then
   $$\begin{aligned} & d(\Psi(\psi)(\mathbf{read}(v))(\theta_1)(\sigma)(\varsigma), \Psi(\psi)(\mathbf{read}(v))(\theta_2)(\sigma)(\varsigma)) \\ = {} & d(\tau \cdot \theta_1(\sigma\{\mathit{first}(\varsigma)/v\})(\mathit{rest}(\varsigma)), \tau \cdot \theta_2(\sigma\{\mathit{first}(\varsigma)/v\})(\mathit{rest}(\varsigma))) \\ = {} & \tfrac{1}{2} \cdot d(\theta_1(\sigma\{\mathit{first}(\varsigma)/v\})(\mathit{rest}(\varsigma)), \theta_2(\sigma\{\mathit{first}(\varsigma)/v\})(\mathit{rest}(\varsigma))) \\ \leq {} & \tfrac{1}{2} \cdot d(\theta_1, \theta_2). \end{aligned}$$
   (c) If $\varsigma \neq \varepsilon$ and $\mathit{first}(\varsigma) = \tau$, then
   $$\begin{aligned} & d(\Psi(\psi)(\mathbf{read}(v))(\theta_1)(\sigma)(\varsigma), \Psi(\psi)(\mathbf{read}(v))(\theta_2)(\sigma)(\varsigma)) \\ = {} & d(\tau \cdot \psi(\mathbf{read}(v))(\theta_1)(\sigma)(\mathit{rest}(\varsigma)), \tau \cdot \psi(\mathbf{read}(v))(\theta_2)(\sigma)(\mathit{rest}(\varsigma))) \\ = {} & \tfrac{1}{2} \cdot d(\psi(\mathbf{read}(v))(\theta_1)(\sigma)(\mathit{rest}(\varsigma)), \psi(\mathbf{read}(v))(\theta_2)(\sigma)(\mathit{rest}(\varsigma))) \\ \leq {} & \tfrac{1}{2} \cdot d(\psi(\mathbf{read}(v))(\theta_1), \psi(\mathbf{read}(v))(\theta_2)) \\ \leq {} & \tfrac{1}{2} \cdot d(\theta_1, \theta_2). \qquad [\psi(\mathbf{read}(v)) \text{ is nonexpansive}] \end{aligned}$$
2. Let $s \equiv \mathbf{fork}(v)$. Then
$$\begin{aligned} & d(\Psi(\psi)(\mathbf{fork}(v))(\theta_1)(\sigma)(\varsigma), \Psi(\psi)(\mathbf{fork}(v))(\theta_2)(\sigma)(\varsigma)) \\ = {} & d((\theta_1(\sigma\{0/v\}))(\theta_1(\sigma\{1/v\})(\varsigma)), (\theta_2(\sigma\{0/v\}))(\theta_2(\sigma\{1/v\})(\varsigma))) \\ \leq {} & \max\{d((\theta_1(\sigma\{0/v\}))(\theta_1(\sigma\{1/v\})(\varsigma)), (\theta_1(\sigma\{0/v\}))(\theta_2(\sigma\{1/v\})(\varsigma))), \\ & \quad d((\theta_1(\sigma\{0/v\}))(\theta_2(\sigma\{1/v\})(\varsigma)), (\theta_2(\sigma\{0/v\}))(\theta_2(\sigma\{1/v\})(\varsigma)))\} \\ & \qquad [\text{ultrametricity}] \\ \leq {} & \max\{d(\theta_1(\sigma\{1/v\})(\varsigma), \theta_2(\sigma\{1/v\})(\varsigma)), d(\theta_1(\sigma\{0/v\}), \theta_2(\sigma\{0/v\}))\} \\ & \qquad [\theta_1(\sigma\{0/v\}) \text{ is nonexpansive}] \\ \leq {} & d(\theta_1, \theta_2). \end{aligned}$$

□

We conclude this section with

DEFINITION 2.3 The denotational semantics $\mathcal{D} : \mathcal{L} \to \mathit{Cont}$ is defined by

$$\mathcal{D}[\![s]\!] = \mathcal{D}(s)(\lambda\sigma \,.\, \lambda\varsigma \,.\, \varepsilon).$$

3. Equivalence Theorem

THEOREM 3.1 *For all $s \in \mathcal{L}$, $\mathcal{O}[\![s]\!] = \mathcal{D}[\![s]\!]$.*

On the way to the proof of this theorem, we first introduce two intermediate semantics.

DEFINITION 3.2 The mapping $\mathcal{H} : \mathit{Res} \to \mathit{Cont}$ is defined by

$$\begin{aligned} \mathcal{H}(\mathrm{E}) &= \lambda\sigma\,.\,\lambda\varsigma\,.\,\varepsilon \\ \mathcal{H}(s:r) &= \mathcal{D}(s)(\mathcal{H}(r)) \end{aligned}$$

The mapping $\mathcal{I} : NRes \to Val_\tau^\infty$ is defined by

$$\begin{aligned} \mathcal{I}(\varsigma) &= \varsigma \\ \mathcal{I}(< r, \sigma, \rho >) &= \mathcal{H}(r)(\sigma)(\mathcal{I}(\rho)) \end{aligned}$$

The following properties of $\mathcal{I}$ are furthermore of importance.

Lemma 3.3 *For all ρ, ρ', and β,*

if $\rho \to_0 \rho'$ then $\mathcal{I}(\rho) = \mathcal{I}(\rho')$, and
if $\rho \xrightarrow{\beta} \rho'$ then $\mathcal{I}(\rho) = \beta \cdot \mathcal{I}(\rho')$.

Proof We only consider a few cases of the proof of the first property.

1. Let $\rho \equiv < (v := e) : r, \sigma, \bar{\rho} >$. Then
$$\begin{aligned} &\mathcal{I}(< (v := e) : r, \sigma, \bar{\rho} >) \\ &= \mathcal{H}((v := e) : r)(\sigma)(\mathcal{I}(\bar{\rho})) \\ &= \mathcal{D}(v := e)(\mathcal{H}(r))(\sigma)(\mathcal{I}(\bar{\rho})) \\ &= \mathcal{H}(r)(\sigma\{\alpha/v\})(\mathcal{I}(\bar{\rho})) \\ &= \mathcal{I}(< r, \sigma\{\alpha/v\}, \bar{\rho} >). \end{aligned}$$
2. Let $\rho \equiv < \mathbf{fork}(v) : r, \sigma, \bar{\rho} >$. Then
$$\begin{aligned} &\mathcal{I}(< \mathbf{fork}(v) : r, \sigma, \bar{\rho} >) \\ &= \mathcal{H}(\mathbf{fork}(v) : r)(\sigma)(\mathcal{I}(\bar{\rho})) \\ &= \mathcal{D}(\mathbf{fork}(v))(\mathcal{H}(r))(\sigma)(\mathcal{I}(\bar{\rho})) \\ &= (\mathcal{H}(r)(\sigma\{0/v\}))(\mathcal{H}(r)(\sigma\{1/v\})(\mathcal{I}(\bar{\rho}))) \\ &= (\mathcal{H}(r)(\sigma\{0/v\}))(\mathcal{I}(< r, \sigma\{1/v\}, \bar{\rho} >)) \\ &= \mathcal{I}(< r, \sigma\{0/v\}, < r, \sigma\{1/v\}, \bar{\rho} >>). \end{aligned}$$

□

The main step in the proof of Theorem 3.1 now follows. Recall that Φ is the higher-order operator used in the definition of $\mathcal{O}$.

Lemma 3.4 $\Phi(\mathcal{I}) = \mathcal{I}$.

Proof We can show that, for all ρ,

$$\Phi(\mathcal{I})(\rho) = \mathcal{I}(\rho)$$

by induction on the complexity of ρ (cf. Definition 1.6). Only a few cases are elaborated on.

1. Let $\rho \equiv < (v := e) : r, \sigma, \rho' >$. Then
$$\begin{aligned} &\Phi(\mathcal{I})(< (v := e) : r, \sigma, \rho' >) \\ &= \Phi(\mathcal{I})(< r, \sigma\{\alpha/v\}, \rho' >) && [< (v := e) : r, \sigma, \rho' > \to_0 < r, \sigma\{\alpha/v\}, \rho' >] \\ &= \mathcal{I}(< r, \sigma\{\alpha/v\}, \rho' >) && [\text{Lemma 1.7, induction}] \\ &= \mathcal{I}(< (v := e) : r, \sigma, \rho' >). && [\text{Lemma 3.3}] \end{aligned}$$
2. Let $\rho \equiv < \mathbf{read}(v) : r, \sigma, \rho' >$. We distinguish three cases.

(a) Assume $\rho' \xrightarrow{\alpha} \rho''$. Then

$$\begin{array}{lll} \Phi(\mathcal{I})(<\mathbf{read}(v) : r, \sigma, \rho' >) & & \\ \quad = \tau \cdot \mathcal{I}(< r, \sigma\{\alpha/v\}, \rho'' >) & & [<\mathbf{read}(v) : r, \sigma, \rho' > \xrightarrow{\tau} < r, \sigma\{\alpha/v\}, \rho'' >] \\ \quad = \mathcal{I}(<\mathbf{read}(v) : r, \sigma, \rho' >). & & [\text{Lemma 3.3}] \end{array}$$

(b) Assume $\rho' \xrightarrow{\tau} \rho''$. Then

$$\begin{array}{lll} \Phi(\mathcal{I})(<\mathbf{read}(v) : r, \sigma, \rho' >) & & \\ \quad = \tau \cdot \mathcal{I}(<\mathbf{read}(v) : r, \sigma, \rho'' >) & & [<\mathbf{read}(v) : r, \sigma, \rho' > \xrightarrow{\tau} <\mathbf{read}(v) : r, \sigma, \rho'' >] \\ \quad = \mathcal{I}(<\mathbf{read}(v) : r, \sigma, \rho' >). & & [\text{Lemma 3.3}] \end{array}$$

(c) Assume ρ' blocks. Then $<\mathbf{read}(v) : r, \sigma, \rho' >$ blocks and hence

$$\Phi(\mathcal{I})(<\mathbf{read}(v) : r, \sigma, \rho' >) = \varepsilon.$$

Since ρ' blocks, $\Phi(\mathcal{I})(\rho') = \varepsilon$. By induction, $\mathcal{I}(\rho') = \varepsilon$. Consequently,

$$\mathcal{I}(<\mathbf{read}(v) : r, \sigma, \rho' >) = \varepsilon.$$

□

We have arrived at the proof of Theorem 3.1:

PROOF Because both $\mathcal{O}$ and $\mathcal{I}$ are fixed point of Φ (Definition 1.5 and Lemma 3.4) and Φ has a unique fixed point, $\mathcal{O}$ and $\mathcal{I}$ are equal. Consequently,

$$\begin{array}{ll} \mathcal{O}[\![s]\!](\sigma)(\varsigma) & \\ \quad = & \mathcal{O}(< s : \mathrm{E}, \sigma, \varsigma >) \\ \quad = & \mathcal{I}(< s : \mathrm{E}, \sigma, \varsigma >) \\ \quad = & \mathcal{H}(s : \mathrm{E})(\sigma)(\mathcal{I}(\varsigma)) \\ \quad = & \mathcal{D}(s)(\mathcal{H}(\mathrm{E}))(\sigma)(\varsigma) \\ \quad = & \mathcal{D}(s)(\lambda\sigma \,.\, \lambda\varsigma \,.\, \varepsilon)(\sigma)(\varsigma) \\ \quad = & \mathcal{D}[\![s]\!](\sigma)(\varsigma). \end{array}$$

□

REFERENCES

[AW85] S.K. Abdali and D.S. Wise. Standard, Storeless Semantics for ALGOL-style Block Structure and Call-by-Name. In A. Melton, editor, *Proceedings of the 1st International Conference on Mathematical Foundations of Programming Semantics*, volume 239 of *Lecture Notes in Computer Science*, pages 1–19, Manhattan, April 1985. Springer-Verlag.

[Ban22] S. Banach. Sur les Opérations dans les Ensembles Abstraits et leurs Applications aux Equations Intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

[Ben82] D.B. Benson. Machine-Level Semantics for Nondeterministic, Parallel Programs. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 15–25, Turin, April 1982. Springer-Verlag.

[BR92] J.W. de Bakker and J.J.M.M. Rutten, editors. *Ten Years of Concurrency Semantics, selected papers of the Amsterdam Concurrency Group.* World Scientific, Singapore, 1992.

[Bru86] A. de Bruin. *Experiments with Continuation Semantics: jumps, backtracking, dynamic networks.* PhD thesis, Vrije Universiteit, Amsterdam, May 1986.

[HSS91] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High-Level Language for PRAMs. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *Proceedings of the 3rd International PARLE Conference*, volume 505 of *Lecture Notes in Computer Science*, pages 304–320, Eindhoven, June 1991. Springer-Verlag.

[KK92] E. Klein and K. Koskimies. How to Pipeline Parsing with Parallel Semantic Analysis. *Structured Programming*, 13(3):99–107, 1992.

[KR90] J.N. Kok and J.J.M.M. Rutten. Contractions in Comparing Concurrency Semantics. *Theoretical Computer Science*, 76(2/3):179–222, 1990.

[MA89] C. McDonald and L. Allison. Denotational Semantics of a Command Interpreter and their Implementation in Standard ML. *The Computer Journal*, 32(5):422–431, October 1989.

[Par83] D. Park. The "Fairness" Problem and Nondeterministic Computing Networks. In J.W. de Bakker and J. van Leeuwen, editors, *Foundations of Computer Science IV, Distributed Systems,* part 2: Semantics and Logic, volume 159 of *Mathematical Centre Tracts*, pages 133–161. Mathematical Centre, Amsterdam, 1983.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Report DAIMI FN-19, Aarhus University, Aarhus, September 1981.

[RS83] J.-C. Raoult and R. Sethi. Properties of a Notation for Combining Functions. *Journal of the ACM*, 30(3):595–611, July 1983.

[RS92] G. Rünger and K. Sieber. A Trace-Based Denotational Semantics for the PRAM-Language FORK. Report 1/1992, Universität des Saarlandes, Saarbrücken, 1992.