Linear Future Semantics and Its Implementation

by

Stefan Kölbl and Mitchell Wand

Computer Science Department

Indiana University

Bloomington, IN 47405

TECHNICAL REPORT NO. 162

Linear Future Semantics and Its Implementation

by

Stefan Kölbl and Mitchell Wand

October, 1984

This material is based on work supported by the National Science Foundation under grant number MCS 83-03325.

Linear Future Semantics and Its Implementation

Stefan Kölbl
Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

This material is based on work supported by the National Science Foundation under grant number MCS 8303325.

Authors' address: Mitchell Wand, Computer Science Department, Brandeis University, Waltham, MA 02254 USA. Stefan Kölbl was killed in a bus accident in Ecuador shortly after completing the first draft of this paper.

Abstract

We describe linear future semantics, an extension of linear history semantics as introduced by Francez, Lehmann, and Pnueli, and show how it can be used to add multiprocessing to languages given by standard continuation semantics. We then demonstrate how the resulting semantics can be implemented. The implementation uses functional abstractions and nondeterminacy to represent the sets of answers in the semantics. We give an example, using a semantic prototyping system based on the language Scheme.

1. Introduction

Linear future semantics is an extension of linear history semantics, as introduced by Francez, Lehmann, and Pnueli in [1], to languages given by a continuation semantics. We suppose that we already have a standard continuation semantics for our language. We want to use it to describe concurrent systems, after only reasonably simple changes in our semantic equations.

The purpose of continuation semantics is to model revocable decisions. Most sequencing decisions in real languages are revocable. For example, in the compound statement $(S_1; S_2)$ the statement S_2 may or may not be executed following S_1, depending on whether S_1 executes an escape or not. Some decisions, however, are irrevocable and should be modelled with a flavor of direct semantics. One example is the production of hard-copy output [2, Sec. 5.1.4.3].

In this paper we are concerned with other kinds of irrevocable decisions, such as the communication between processes in a concurrent system, the termination of processes, and the passage of time. We may describe the meaning of a process by means of the irrevocable decisions it makes during the computation. The result is a set of sequences of irrevocable decisions associated with each process. This set contains all sequences which may occur if the process is executed. As in [1], we will call this the set of *communication sequences* of a process.

In [1], the semantics kept track of the possible sequences of decisions the passage of some unit of time without a communication was marked by the symbol δ (a "tick") in the communication sequence. In addition, each operation which took time introduced an uncompleted history (the symbol $\perp$) into the set of sequences. As a result, processes which never communicated still made a contribution to the set of sequences. Consequently, no ordering on communication sequences was needed, and the power set ordering on sets of sequences sufficed. We will use a similar device.

A direct semantics was used in [1], so that a set of sequences was associated with each command: the set of communication *histories* of that command. Our contribution is to extend this idea to continuation semantics, by making the set of communication sequences be the set of answers of the continuations. Thus a continuation represents a set of possible communication *futures*. The semantics of a command is a transformation on sets of possible futures.

Given the set of communication sequences for every participating process in a concurrent system, we will show how to combine these sets in order to give a description of the whole system, using a coalesced merge as in [1]. This operation models the irrevocable decisions of process scheduling, *etc.*, so it is done in direct semantics, operating on the answers from the continuation semantics.

In section 2, we discuss linear future semantics in general. We then apply it to a pre-existing base language with continuation semantics, given in Section 3. Section 4 shows the required changes to the semantic equations. In Section 5, we show how the language can be implemented, using a semantic prototyping system based on the language Scheme [4]. The implementation uses functional abstractions (suspensions) to model the sets of sequences.

2. Linear Future Semantics

2.1 Linear Future Semantics for a Single Process

In this paper we will deal only with a concurrent system of two processes, communicating explicitly via messages. Let H be a domain of communicable

values, P a set of process names and *Msg* a set of messages. The set of *communications of a process* is the set

$$D_P = \{\,\textbf{tick}\,\} \cup \{\,\textbf{send}\ b \mid b \in H\,\} \cup \{\,\textbf{receive}\ b \mid b \in H\,\} \cup \{\,\textbf{output}\ b \mid b \in H\,\}$$

The set of *terminators of a process* is the set

$$T_P = \{\,\textbf{proc-termin}\ msg \mid msg \in Msg\,\} \cup \{\,\textbf{not-done-yet},\ \textbf{waiting-to-send},\ \textbf{waiting-to-receive}\,\}$$

The set of *communication sequences of a process* is the set

$$Q_P = D_P^* \times T_P.$$

Q_P consists only of finite sequences; the presence of infinite behaviors will be deduced from the presence of unbounded sets of finite sequences, as in [1]. Let 2^{Q_P} denote the powerset of Q_P. Then 2^{Q_P} is a semantic domain ordered under subset inclusion. The *linear future semantics for a single process* is a mapping

$$\mathcal{P} : \langle Pgm \rangle \to \langle input\text{-}states \rangle \to P \to 2^{Q_P}.$$

If we apply the linear future semantics of a single process to an input state and a process name, we receive the set of communication sequences which can result if the process is executed in an arbitrary environment.

We now take a closer look at the communications and terminators. We assume that we are in a continuation semantics framework, and we adopt the general principle that the answer produced by a continuation is the set of possible communications sequences for this process, that is, the set of possible futures. Inside a process, therefore, a continuation will be of type $V \to 2^{Q_P}$. A programming language phrase corresponds to a transformer on continuations. Let κ denote a continuation waiting for a result, and consider some of the plausible operations on κ and how they correspond to the elements of D_P.

1. The action of the program might not include any communication, but it will consume some amount of time. A **tick** (the δ in [1]) denotes the passage of time. We have to include enough ticks so that every loop produces at least one tick. This leads to the desired property that long computations result in long communication sequences, no matter if the process is actually communicating or not, and therefore least fixed points work correctly as in [1]. We associate a tick with every time we

pass a value v to the continuation, unless a communication takes place. The future of the process is therefore the set

$$\{\,\mathbf{tick} \bullet \alpha \mid \alpha \in \kappa v\,\}.$$

2. The process might communicate with the outside world by outputting a value. We assume that this communication cannot fail and that it does not require synchronization. The communication **output** b denotes a communication with the outside world. Every time the process executes an output command with a value b we prefix **output** b to the future of the rest of the computation, *i.e.* the future is the set

$$\{\,\mathbf{output}\ b \bullet \alpha \mid \alpha \in \kappa b^*\,\}$$

where b^* is the value passed to its continuation by the output command.

3. The process may wish to send a value to another process. This communication forces synchronization and it can also fail. If we execute a send-command with a value b, the future of the process will be the set

$$\{\,\mathbf{send}\ b \bullet \alpha \mid \alpha \in \kappa b^*\,\}$$

where b^* is the value produced by the **send** command, plus the singleton set $\{\,\mathbf{waiting\text{-}to\text{-}send}\,\}$ to denote a possible failure.

4. Similarly, the process may wish to receive a value from another process. In this case the result is more complex. Because we want to give an independent semantics for each process, we do not know which value b is received. To include all possible cases the future has to be the union of

$$\bigcup_{b \in H} \{\,\mathbf{receive}\ b \bullet \alpha \mid \alpha \in \kappa b\,\}$$

and the singleton set $\{\,\mathbf{waiting\text{-}to\text{-}receive}\,\}$.

5. The terminator **not-done-yet** has the same function as the symbol $\perp$ in [1]. Since the passage of time is a one of the factors in determining the semantics of a process, we have to include all incomplete communication sequences which occur during the computation. That means that every time we append something to the communication sequence, we also introduce a future consisting of the singleton set $\{\,\mathbf{not\text{-}done\text{-}yet}\,\}$. Consequently, as in [1], no ordering is needed on communication sequences and the power set ordering on 2^{Q_P} suffices.

6. The singleton set $\{\,\mathbf{proc\text{-}termin}\ msg\,\}$ is the future of a process which has terminated.

2.2 The Linear Future Semantics of a System

We now look at a system of two communicating processes. Let H, P, Msg be as in the previous section. The set of *communications of a system* is

$$D_S = \{\,\textbf{tick}\,\} \cup \{\,\textbf{output}\ b \mid b \in H\,\}$$

and the set of *terminators of a system* is

$$T_S = \{\,\textbf{sys-termin}\ msg \mid msg \in Msg\,\} \cup \{\,\textbf{not-done-yet}, \textbf{deadlocked}\,\}.$$

The set of *communication sequences of a system* is

$$Q_S = D_S^* \times T_S$$

Again, 2^{Q_S} is the semantic domain of sets of communication sequences.

{ **deadlocked** } is the future of a system which ran into a deadlock. (In [1], this was denoted by $e(\emptyset)$). This can happen either if both processes are waiting for each other, or if one process is waiting for a communication while the other one has already terminated. { **sys-termin** *msg* } is analogous to { **proc-termin** *msg* } and is the future of a system whose processes have all terminated, and { **not-done-yet** } denotes an incomplete computation.

We are now ready to define the "binding" operator *merge*, which performs a coalesced merge on two sets of communication sequences. It is based on the function *filter*, which takes two single communication sequences and produces a set consisting of all the possible ways in which the two sequences may be combined consistently.

$$merge : 2^{Q_S} \times 2^{Q_S} \rightarrow 2^{Q_P}$$

$$merge\,(Q_1, Q_2) = \bigcup_{\substack{\eta_1 \in Q_1 \\ \eta_2 \in Q_2}} filter\,(\eta_1, \eta_2)$$

where η_1 and η_2 are single communication sequences. The definition of *filter* is

$$\begin{aligned}
&filter\,(\eta_1, \eta_2) = \\
&\qquad [\textbf{if}\ first\,(\eta_1) \in \text{Nonsynch} \\
&\qquad\quad \textbf{then}\ \{\, first\,(\eta_1) \bullet \alpha \mid \alpha \in filter\,(rest\,(\eta_1), \eta_2)\,\}\,] \\
&\qquad \cup [\textbf{if}\ first\,(\eta_2) \in \text{Nonsynch} \\
&\qquad\quad \textbf{then}\ \{\, first\,(\eta_2) \bullet \alpha \mid \alpha \in filter\,(\eta_1, rest\,(\eta_2))\,\}\,] \\
&\qquad \cup [\textbf{if}\ (\exists b)((first\,(\eta_1) = \textbf{send}\ b\ \textbf{and}\ first\,(\eta_2) = \textbf{receive}\ b) \\
&\qquad\qquad \textbf{or}\ (first\,(\eta_1) = \textbf{receive}\ b\ \textbf{and}\ first\,(\eta_2) = \textbf{send}\ b)) \\
&\qquad\quad \textbf{then}\ \{\, \textbf{tick} \bullet \alpha \mid \alpha \in filter\,(rest\,(\eta_1), rest\,(\eta_2))\,\}\,] \\
&\qquad \cup [\textbf{if}\ length\,(\eta_1) = 1\ \textbf{and}\ length\,(\eta_2) = 1 \\
&\qquad\quad \textbf{then}\ filter\text{-}terminator\,(\eta_1, \eta_2)\,]
\end{aligned}$$

	ndy	*wts*	*wtr*	*proc-term* msg_1
ndy	{ *ndy* }	{ *ndy* }	{ *ndy* }	{ *ndy* }
wts	{ *ndy* }	{ *dlk* }	{ *ndy* }	{ *dlk* }
wtr	{ *ndy* }	{ *ndy* }	{ *dlk* }	{ *dlk* }
proc-term msg_2	{ *ndy* }	{ *dlk* }	{ *dlk* }	{ *sys-term* msg^* }

Legend: *ndy* = **not-done-yet**, *wts* = **waiting-to-send**, *wtr* = **waiting-to-receive**, *proc-term msg* = **proc-termin** *msg*, *dlk* = **deadlocked**.

Table 1. Definition of *filter-terminator*

Here, as in [1], we assume an "**else** $\emptyset$" appended to each clause, and we let Nonsynch = { **tick** } ∪ { **output** b | $b \in H$ } be the subset of D_P which contains only the elements that do not force synchronization. In this definition, if either sequence starts with such an element, that element may occur first. If both sequences start with a matching synchronizing communication, then the outside world sees only a **tick**. If both sequences consist only of a terminator, then more care is required. In any other case, if the sequences disagree on the value of the transmission or if only one sequence consists only of a terminator, the pair is regarded as inconsistent and does not contribute anything to the answer. Because all sequences are finite, the recursion in *filter* poses no difficulties.

filter-terminator is defined in Table 1. It may be regarded as a transcript of $\sigma_1 \times \sigma_2$ in [1] for the case of two processes. The message msg^* may be a combined message of msg_1 and msg_2.

3. The Base Language

Our base language is a simple expression language with input and output, but without multiprocessing. In the next section we will add multiprocessing and show the changes in the semantic equations.

The language is described in Table 2–Table 6. It is an off-the-shelf example we have used for several years to illustrate continuation semantics. It is an expression language which manipulates Lisp S-expressions, has functions as first-class citizens, does input and output, and has escapes (to the top level only). It is not intended to be minimal, but using this non-minimal example will help illustrate how our techniques can apply to pre-existing languages.

The presentation is relatively standard, except for the function *casefn* to choose between alternatives in a disjoint union. $casefn : (A + B) \to (A \to C) \to (B \to C) \to C$ may be defined as:

$$casefn = \lambda x f g.(x \in A) \to f(x\ \mathtt{A}),\ \mathtt{g(x}\ B)$$

$\langle Pgm \rangle$ ::= $\langle Exp \rangle$
$\langle Exp \rangle$ ::= $\langle Const \rangle$
$\langle Exp \rangle$ ::= $\langle Ident \rangle$
$\langle Exp \rangle$ ::= (**error** $\langle Msg \rangle$)
$\langle Exp \rangle$ ::= (**if** $\langle Exp \rangle$ **then** $\langle Exp \rangle$ **else** $\langle Exp \rangle$)
$\langle Exp \rangle$ ::= (**fn** $\langle Ident \rangle \langle Exp \rangle$)
$\langle Exp \rangle$ ::= (**rec** $\langle Ident \rangle$(**fn** $\langle Ident \rangle \langle Exp \rangle$))
$\langle Exp \rangle$ ::= ($\langle Exp \rangle \langle Exp \rangle$)
$\langle Exp \rangle$ ::= (**read**)
$\langle Exp \rangle$ ::= (**print** $\langle Exp \rangle$)
$\langle Const \rangle$ ::= $\langle Number \rangle$
$\langle Const \rangle$ ::= $\langle quoted\ S\text{-}expr \rangle$
$\langle Const \rangle$ ::= **t** | **nil** | **car** | **cdr** | **eq** | **atom** | **cons**
$\langle Ident \rangle$ ::= $\langle any\ other\ atom \rangle$

Table 2. Syntax of the base language.

Name	Definition	Description
Bas	= S-expressions	Basic Values
S	$= Bas^* \times Bas^*$	States (input, output)
Msg		Messages
A	$= Bas^* \times Msg$	Answers
E	$= Bas + F$	Expressed Values
Ident		Identifiers
Env	$= Ident \rightarrow E$	Environments
K	$= E \rightarrow S \rightarrow A$	Expression Continuations
F	$= E \rightarrow K \rightarrow S \rightarrow A$	Functional Values

Table 3. Domains for the base language.

$$\mathcal{P} : \langle Pgm \rangle \rightarrow Bas^* \rightarrow A$$
$$\mathcal{E} : \langle Exp \rangle \rightarrow Env \rightarrow K \rightarrow S \rightarrow A$$
$$\mathcal{C} : \langle Const \rangle \rightarrow E$$

Table 4. Valuations for the base language

4. Adding Multiprocessing

To add multiprocessing to this base language, we first change the syntax

$$\mathcal{P}[\![\langle Exp\rangle]\!] = \lambda w.\mathcal{E}[\![\langle Exp\rangle]\!](\lambda I.\text{“unbound”})(\lambda v.terminate[\![\text{“termin”}]\!])(mk\text{-}state\ w\ empty)$$

$$\mathcal{E}[\![\langle Const\rangle]\!] = \lambda\rho\kappa.\kappa(\mathcal{C}[\![\langle Const\rangle]\!])$$

$$\mathcal{E}[\![\langle Ident\rangle]\!] = \lambda\rho\kappa.\rho(\langle Ident\rangle) = \text{“unbound”} \rightarrow terminate[\![\text{“unbound id”}]\!],\ \kappa(\rho(\langle Ident\rangle))$$

$$\mathcal{E}[\![(\mathbf{error}\langle msg\rangle)]\!] = \lambda\rho\kappa.terminate[\![\text{“Error”}\langle msg\rangle]\!]$$

$$\mathcal{E}[\![(\mathbf{if}\ \langle Exp\rangle_0\ \mathbf{then}\ \langle Exp\rangle_1\ \mathbf{else}\ \langle Exp\rangle_2)]\!] = \lambda\rho\kappa.\mathcal{E}[\![\langle Exp\rangle_0]\!]\,\rho\,(\lambda\beta.\,\beta \neq nil \rightarrow \mathcal{E}[\![\langle Exp\rangle_1]\!]\rho\kappa,\ \mathcal{E}[\![\langle Exp\rangle_2]\!]\rho\kappa)$$

$$\mathcal{E}[\![(\mathbf{fn}\ \langle Ident\rangle\ \langle Exp\rangle)]\!] = \lambda\rho\kappa.\,\kappa(inE(\lambda v\kappa_1.\,\mathcal{E}[\![\langle Exp\rangle]\!]\rho[v/\langle Ident\rangle]\kappa_1))$$

$$\mathcal{E}[\![(\mathbf{rec}\ \langle Ident\rangle_1\ (\mathbf{fn}\ \langle Ident\rangle_2\ \langle Exp\rangle))]\!] =$$
$$\lambda\rho\kappa.\kappa\,(fix(\lambda\theta.\,inE(\lambda v\kappa_1.\mathcal{E}[\![\langle Exp\rangle]\!]\rho[\theta/\langle Ident\rangle_1][v/\langle Ident\rangle_2]\kappa_1)))$$

$$\mathcal{E}[\![(\langle Exp\rangle_1\ \langle Exp\rangle_2)]\!] = \lambda\rho\kappa.\mathcal{E}[\![\langle Exp\rangle_1]\!]\rho(\lambda v.$$
$$casefn\ v\ (\lambda b.terminate[\![\text{“attempt to apply non fcn”}]\!]\ )(\lambda f.\mathcal{E}[\![\langle Exp\rangle_2]\!]\rho(\lambda a.f\ a\ \kappa)))$$

$$\mathcal{E}[\![(\mathbf{read})]\!] = \lambda\rho.do\text{-}read$$

$$\mathcal{E}[\![(\mathbf{print}\langle Exp\rangle)]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![\langle Exp\rangle]\!]\rho(do\text{-}print\,\kappa)$$

$$\mathcal{C}[\![\mathbf{car}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.\kappa\,(inE(car\ b))\ )(\lambda f.terminate[\![\text{“can't take car of fcn”}]\!]))$$

$$\mathcal{C}[\![\mathbf{cdr}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.\kappa\,(inE(cdr\ b))\ )(\lambda f.terminate[\![\text{“can't take cdr of fcn”}]\!]))$$

$$\mathcal{C}[\![\mathbf{atom}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.\kappa\,(inE(atom\ b))\ )(\lambda f.terminate[\![\text{“can't take atom of fcn”}]\!]))$$

$$\mathcal{C}[\![\mathbf{eq}]\!] = inE(\lambda v_1\,\kappa_1.casefn\ v_1$$
$$(\lambda b_1.\kappa_1\,(inE(\lambda v_2\kappa_2.casefn\ v_2$$
$$(\lambda b_2.\kappa_2\,(inE(eq\ b_1\ b_2)))$$
$$(\lambda f.terminate[\![\text{“can't eq fcn”}]\!])))$$
$$(\lambda f.terminate[\![\text{“can't eq fcn”}]\!]))$$

$$\mathcal{C}[\![\mathbf{cons}]\!] = inE(\lambda v_1\,\kappa_1.casefn\ v_1$$
$$(\lambda b_1.\kappa_1\,(inE(\lambda v_2\kappa_2.casefn\ v_2$$
$$(\lambda b_2.\kappa_2\,(inE(cons\ b_1\ b_2)))$$
$$(\lambda f.terminate[\![\text{“can't cons fcn”}]\!])))$$
$$(\lambda f.terminate[\![\text{“can't cons fcn”}]\!]))$$

$$\mathcal{C}[\![\mathbf{t}]\!] = inE(t)$$

etc.

Table 5. Semantic Equations for the Base Language

$mk\text{-}state = \lambda w_1\ w_2.\langle w_1, w_2\rangle$
$do\text{-}read = \lambda\kappa\sigma.null?\,\sigma_1 \rightarrow terminate[\![\text{"eof on read"}]\!]\sigma,\ \kappa\,(first\,\sigma_1)(mk\text{-}state\,(rest\,\sigma_1)\,\sigma_2)$
$do\text{-}print = \lambda\kappa v\sigma.casefn\ v\ (\lambda b.\kappa v(mk\text{-}state\,\sigma_1(\sigma_2||b))\ )(\lambda f.terminate[\![\text{"can't print fcn"}]\!]\sigma)$
$terminate = \lambda msg\,v\sigma.\,(\sigma_2||msg)$
$casefn = \lambda v f_1 f_2.\,v \in Bas \rightarrow f_1\,(v|Bas),\ f_2\,(v|F)$

Table 6. Auxiliaries for the Base Language

to add the new nonterminal $\langle Sys\rangle$ with the production

$$\langle Sys\rangle\ ::=\ (\ \textbf{parbegin}\ \langle Pgm\rangle\ \langle Pgm\rangle)$$

and to add productions for the communication primitives:

$$\langle Exp\rangle ::= (\ \textbf{receive}\ )$$
$$\langle Exp\rangle ::= (\ \textbf{send}\ \langle Exp\rangle)$$

We now address ourselves to the problem of defining the semantics of the altered language. We assume the processes have independent local states and communicate only via sending and receiving messages. We should be able to give the new semantics with only few changes in our equations, and still describe the concurrent system in a sufficient way. The semantics of a program will now be a mapping from two input-states (one for each process) into a set of communication sequences. The associated semantic domains are 2^{Q_P} and 2^{Q_S}, as shown in section 2.

The semantics domains *Bas*, *Msg*, *Ident*, *S*, and *Env* are the same as in the base language. To these we add the following:

H	$=$	Bas	Communicable Values
P	$=$	$\{1, 2\}$	Process Names
A_S	$=$	2^{Q_S}	System Answers
A_P	$=$	$P \rightarrow 2^{Q_P}$	Process Answers
K	$=$	$E \rightarrow S \rightarrow A_P$	Expression Continuations
F	$=$	$E \rightarrow K \rightarrow S \rightarrow A_P$	Functional Values

We included the set P of process names. Our example is so simple that it would be possible to dispense with process names entirely. Nonetheless, considering a possible extension to a system of more than two processes, we chose this place to add them. By associating P with the answers A_P we

relieve most of the equations of the need to deal with process names. The valuations are now

$$\begin{aligned}
&\mathcal{S} : \langle Sys \rangle \to S \to S \to A_S \\
&\mathcal{P} : \langle Pgm \rangle \to S \to A_P \\
&\mathcal{E} : \langle Exp \rangle \to Env \to K \to S \to A_P \\
&\mathcal{C} : \langle Const \rangle \to E
\end{aligned}$$

We next modify the semantic equations to produce these sets of communication sequences, in accordance with the protocol in Section 2.

1. We add the new top level

$$\begin{aligned}
&\mathcal{S}[\![(\ \textbf{parbegin}\ \langle Pgm \rangle_1 \langle Pgm \rangle_2)]\!] \\
&\quad = \lambda w_1 w_2 . merge(\mathcal{P}[\![\langle Pgm \rangle_1]\!] w_1 1,\ \mathcal{P}[\![\langle Pgm \rangle_2]\!] w_2 2)
\end{aligned}$$

Note that this is a direct semantics, since *merge* deals with the answers from the two processes. We also add two new equations for our communication primitives:

$$\begin{aligned}
\mathcal{E}[\![(\ \textbf{send}\ \langle Exp \rangle)]\!] &= \lambda \rho \kappa . \mathcal{E}[\![\langle Exp \rangle]\!] \rho (do\text{-}send\, \kappa) \\
\mathcal{E}[\![(\ \textbf{receive}\ )]\!] &= \lambda \rho \kappa . do\text{-}receive\, \kappa
\end{aligned}$$

where *do-send* and *do-receive* are defined in Table 8.

2. To define the interface between the processes and the outside world, assume that the processes read from separate input streams (part of their local state), but write to a single shared output stream. Thus the **print** expression is associated with the **output** action of Section 2. Hence the second component of the state domain (used for maintaining the output stream) is no longer necessary. We do not alter it, however, in order to minimize the changes to the semantics.

3. Every appearence of κv for some continuation κ and some value v is replaced by $(do\text{-}output\, \kappa\, v)$ inside the *do-print*-routine and by $do\text{-}tick\, \kappa\, v$ elsewhere, where *do-output* and *do-tick* are defined in Table 8.

4. The terminator is now defined as

$$terminate = \lambda msg\, v \sigma \pi . \{\ \textbf{proc-termin}\ msg\}$$

The definition of *merge* is the same as in Section 2.3.

This is a complete list of the necessary modifications. For an extension to a system of more than two processes, *do-send* and *do-receive* would include the process names, and *merge* would match them. Table 7 gives a complete

list of the semantic equations of the concurrent system. Table 8 gives a list of the auxiliaries.

5. The Implementation

Our final goal is to implement our language in a natural and correctness-preserving way. We use for our implementation language the semantic prototyping system described in [4]. This is a suite of tools based on Scheme 84 [3]. Scheme is a dialect of Lisp with lexical scoping and functions as first-class citizens. Though its reduction is applicative order rather than leftmost, it is adequate for modelling the reductions of continuation semantics in the lambda-calculus. Added to this base is a type-checker and a syntax-directed transducer generator, which permit a rapid transcription of semantic equations.

As with the semantics, we started off with a pre-existing implementation of the base language, and our goal was to implement the modified semantics with as few changes as possible.

The major decisions for the implementation are the choice of representations for the various semantic domains. In general, we need not represent every possible value in a domain; we need only represent those that are reachable in the course of a computation. This often simplifies the representations.

The most crucial decision is the representation of the sets of communication sequences. For the implementation, we are not interested in the set of all possible communication sequences as a result of a computation. We are interested instead in choosing *one* of those possible results. We are also not concerned with incomplete computation sequences. Their purpose was to establish a simple ordering on our domains. Since we know that this ordering exists and that we can define the least fixpoint, we can confine ourselves to a single complete sequence as an answer. This sequence has to be chosen out of the set of all possible sequences. We also want to ensure that we make a *fair choice*, *i.e.* that every complete communication sequence has the same chance to be selected.

These considerations lead us to the following representation decisions:

1. We can ignore sequences ending with **not-done-yet**. Incomplete sequences are represented by an actual "not-done-yet", *i.e.* our computation is not yet finished.
2. We can also ignore sequences ending with **waiting-to-send** or **waiting-to-receive**. We represent these sequences with real waiting, *i.e.* in *merge* we only process sequences which are not headed by a send or receive communication, unless we encounter a matching pair of communications. We get a deadlock if either both sequences are headed

$$\mathcal{S}[\![(\textbf{parbegin}\ \langle Pgm\rangle_1\langle Pgm\rangle_2)]\!] = \lambda w_1 w_2.merge(\mathcal{P}[\![\langle Pgm\rangle_1]\!]w_1 1,\ \mathcal{P}[\![\langle Pgm\rangle_2]\!]w_2 2).$$

$$\mathcal{P}[\![\langle Exp\rangle]\!] = \lambda w.\mathcal{E}[\![\langle Exp\rangle]\!](\lambda I.\text{“unbound”})(\lambda v.terminate[\![\text{“termin”}]\!])(mk\text{-}state\ wempty)$$

$$\mathcal{E}[\![\langle Const\rangle]\!] = \lambda\rho\kappa.do\text{-}tick\ \kappa\ (\mathcal{C}[\![\langle Const\rangle]\!])$$

$$\mathcal{E}[\![\langle Ident\rangle]\!] = \lambda\rho\kappa.\rho(\langle Ident\rangle) = \text{“unbound”} \rightarrow terminate[\![\text{“unbound id”}]\!],\ do\text{-}tick\ \kappa\ (\rho(\langle Ident\rangle))$$

$$\mathcal{E}[\![(\textbf{error}\langle msg\rangle)]\!] = \lambda\rho\kappa.terminate[\![\text{“Error”}\langle msg\rangle]\!]$$

$$\mathcal{E}[\![(\textbf{if}\ \langle Exp\rangle_0\textbf{then}\ \langle Exp\rangle_1\textbf{else}\ \langle Exp\rangle_2)]\!] = \lambda\rho\kappa.\mathcal{E}[\![\langle Exp\rangle_0]\!]\,\rho\,(\lambda\beta.\beta \rightarrow \mathcal{E}[\![\langle Exp\rangle_1]\!]\rho\kappa,\ \mathcal{E}[\![\langle Exp\rangle_2]\!]\rho\kappa)$$

$$\mathcal{E}[\![(\textbf{fn}\ \langle Ident\rangle\ \langle Exp\rangle)]\!] = \lambda\rho\kappa.do\text{-}tick\ \kappa\ (inE(\lambda v\kappa_1.\ \mathcal{E}[\![\langle Exp\rangle]\!]\rho[v/\langle Ident\rangle]\kappa_1))$$

$$\mathcal{E}[\![(\textbf{rec}\ \langle Ident\rangle_1\ (\textbf{fn}\ \langle Ident\rangle_2\ \langle Exp\rangle))]\!] =$$
$$\lambda\rho\kappa.\ do\text{-}tick\ \kappa\ (fix(\lambda\theta.\ inE(\lambda v\kappa_1.\ \mathcal{E}[\![\langle Exp\rangle]\!]\rho[\theta/\langle Ident\rangle_1][v/\langle Ident\rangle_2]\kappa_1)))$$

$$\mathcal{E}[\![(\langle Exp\rangle_1\ \langle Exp\rangle_2)]\!] = \lambda\rho\kappa.\mathcal{E}[\![\langle Exp\rangle_1]\!]\rho(\lambda v.$$
$$casefn\ v\ (\lambda b.terminate[\![\text{“attempt to apply non fcn”}]\!]\ )(\lambda f.\mathcal{E}[\![\langle Exp\rangle_2]\!]\rho(\lambda a.f\ a\ \kappa)))$$

$$\mathcal{E}[\![(\textbf{read})]\!] = \lambda\rho.do\text{-}read$$

$$\mathcal{E}[\![(\textbf{print}\ \langle Exp\rangle)]\!] = \lambda\rho\kappa.\ \mathcal{E}[\![\langle Exp\rangle]\!]\rho(do\text{-}print\ \kappa)$$

$$\mathcal{E}[\![(\textbf{send}\ \langle Exp\rangle)]\!] = \lambda\rho\kappa.\ \mathcal{E}[\![\langle Exp\rangle]\!]\rho(do\text{-}send\ \kappa)$$

$$\mathcal{E}[\![(\textbf{receive})]\!] = \lambda\rho\kappa.\ do\text{-}receive\ \kappa.$$

$$\mathcal{C}[\![\textbf{car}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.do\text{-}tick\ \kappa\ (inE(car\ b))\ )(\lambda f.terminate[\![\text{“can't take car of fcn”}]\!]))$$

$$\mathcal{C}[\![\textbf{cdr}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.do\text{-}tick\ \kappa\ (inE(cdr\ b))\ )(\lambda f.terminate[\![\text{“can't take cdr of fcn”}]\!]))$$

$$\mathcal{C}[\![\textbf{atom}]\!] = inE(\lambda v\kappa.$$
$$casefn\ v\ (\lambda b.do\text{-}tick\ \kappa\ (inE(atom\ b))\ )(\lambda f.terminate[\![\text{“can't take atom of fcn”}]\!]))$$

$$\mathcal{C}[\![\textbf{eq}]\!] = inE(\lambda v_1\ \kappa_1.casefn\ v_1$$
$$(\lambda b_1.do\text{-}tick\ \kappa_1\ (inE(\lambda v_2\kappa_2.casefn\ v_2$$
$$(\lambda b_2.do\text{-}tick\ \kappa_2\ (inE(eq\ b_1\ b_2)))$$
$$(\lambda f.terminate[\![\text{“can't eq fcn”}]\!])))$$
$$(\lambda f.terminate[\![\text{“can't eq fcn”}]\!]))$$

$$\mathcal{C}[\![\textbf{cons}]\!] = inE(\lambda v_1\ \kappa_1.casefn\ v_1$$
$$(\lambda b_1.do\text{-}tick\ \kappa_1\ (inE(\lambda v_2\kappa_2.casefn\ v_2$$
$$(\lambda b_2.do\text{-}tick\ \kappa_2\ (inE(cons\ b_1\ b_2)))$$
$$(\lambda f.terminate[\![\text{“can't cons fcn”}]\!])))$$
$$(\lambda f.terminate[\![\text{“can't cons fcn”}]\!]))$$

$$\mathcal{C}[\![\textbf{t}]\!] = inE(t)$$

etc.

Table 7. Equations modified for multiprocessing

$$mk\text{-}state = \lambda w_1\, w_2.\langle w_1, w_2\rangle$$
$$do\text{-}read = \lambda\kappa\sigma.null?\,\sigma_1 \rightarrow terminate[\![\text{"eof on read"}]\!]\sigma,\ do\text{-}tick\,\kappa\,(first\,\sigma_1)(mk\text{-}state\,(rest\,\sigma_1)\,\sigma_2)$$
$$do\text{-}print = \lambda\kappa v\sigma.casefn\ v\ (\lambda b.do\text{-}output\ \kappa\ v\ (mk\text{-}state\,\sigma_1(\sigma_2\|b))\ )(\lambda f.terminate[\![\text{"can't print fcn"}]\!]\sigma)$$
$$terminate = \lambda msg\,v\sigma\pi.\{\,\textbf{proc-termin}\ msg\,\}$$
$$do\text{-}tick = \lambda\kappa v\sigma\pi.\{\,\textbf{tick} \bullet \alpha \mid \alpha \in \kappa v\sigma\pi\,\} \cup \{\,\textbf{not-done-yet}\,\}$$
$$do\text{-}output = \lambda\kappa v\sigma\pi.\{\,\textbf{output}\ v \bullet \alpha \mid \alpha \in \kappa v\sigma\pi\,\} \cup \{\,\textbf{not-done-yet}\,\}$$
$$do\text{-}send = \lambda\kappa\sigma\pi.casefn\ v(\lambda b.\{\,\textbf{send}\ b \bullet \alpha \mid \alpha \in \kappa v\sigma\pi\,\} \cup \{\,\textbf{waiting-to-send, not-done-yet}\,\})\ (\lambda f.\ terminate[\![\text{"can't send fcn"}]\!])$$
$$do\text{-}receive = \lambda\kappa\sigma\pi.\bigcup_{b\in H}\{\,\textbf{receive}\ b \bullet \alpha \mid \alpha \in \kappa b\sigma\pi\,\} \cup \{\,\textbf{waiting-to-receive, not-done-yet}\,\}$$
$$casefn = \lambda v f_1 f_2.\, v \in Bas \rightarrow f_1\,(v \mid Bas),\ f_2\,(v \mid F)$$

Table 8. Auxiliaries for multiprocessing semantics

by a send communication, or both are headed by a receive communication. We will also get a deadlock if one process has terminated whereas the other one still has a communication sequence headed by a send- or receive- communication.

3. Whenever we have a set of sequences representing possible futures, we will *suspend* the choice until it is needed. Hence, a continuation, instead of being a function $V \rightarrow 2^{Q_S}$, becomes a function $V \rightarrow Q_S$ which performs the selection (perhaps implicitly). For example, if we have a set of sequences associated with a receive-command, we always choose only one sequence out of this set by waiting for the value b of the matching **send** b. The representation of the set

$$\bigcup_{b\in H}\{\,\textbf{receive}\ b \bullet \alpha \mid \alpha \in \kappa b\sigma\pi\,\}$$

becomes the function

$$(\lambda b.\kappa' b\sigma\pi),$$

where κ' is the representation of κ that makes the choice.

We now give the description of the representations, and show how they are translated into the language of the Scheme 84 transducer generator. In many cases we have used a fairly coarse representation, in that not every element of the representation corresponds to an element of the represented type.

$T_P = \{\textbf{proc-termin}\ msg \mid msg \in Msg\}$
`(define-type-abbrev proc-termin msg)`

$D_S = \{\textbf{tick}\} \cup \{\textbf{output}\ b \mid b \in Bas\}$
`(define-type-abbrev sys-comm (heterogeneous))`

In the implementation, D_S is represented as `(heterogeneous)`, as this information need not be type-checked.

$A_S = 2^{Q_S}$
`(define-type-abbrev sys-answer (list sys-commun))`

As discussed above, terminators are not represented, and sets of possible sequences are represented by the chosen sequence.

$A_P = P \rightarrow 2^{Q_P}$
`(define-type-abbrev proc-answer (-> (seq procname) proc-commun-seq))`

This brings us to the key data type, that of process communication sequences. We use the domain $(D_P + T_P)^*$ to represent these sequences. The equations for the types in the representation are:

$$Q_P = (D_P + T_P)^*$$
$$D_P = D_S + C_r + C_s$$
$$C_r = Bas \rightarrow A_P$$
$$C_s = Bas$$

```
(define-type-abbrev proc-commun-seq (list proc-commun))

(define-type-abbrev proc-commun
   (quadrunion
      sys-commun
      (-> (seq bas) proc-answer)
      bas
      proc-termin))
```

Here, `quadrunion` is a type constructor that constructs a four-way union type, with a four-way discriminator *casecom* similar to *casefn* and injection functions `inP1`, `inP2`, `inP3`, and `inP4`.

Thus, a `proc-commun` is either (1) a `sys-commun`, corresponding to a non-synchronizing element, (2) a function from communicable values (`bas`) to `proc-answer`, corresponding to a receive communication, (3) a basic value, corresponding to a send communication, or (4) a terminator. If the first element of a `proc-commun-seq` comes from the second or fourth components,

then the remainder of the representing sequence is ignored, as the receive-function encodes the rest of the represented sequence, and a terminator of course terminates the sequence. This is an instance of the coarseness mentioned previously; here we trade off help from the type-checker against freedom of choice of representation.

The new auxiliaries are now

$$
\begin{aligned}
\textit{do-output} &= \lambda\kappa b\sigma\pi.\textit{inD}_P\ (\textbf{output}\ b) \bullet \kappa\,(\textit{inE}(b))\sigma\pi \\
\textit{do-send} &= \lambda\kappa v\sigma\pi.\textit{casefn}\ v \\
&\qquad (\lambda b.\textit{inD}_P\ (b) \bullet \kappa v\sigma\pi) \\
&\qquad (\lambda f.\textit{terminate}[\![\ \text{“}\textit{can't send function}\text{”}]\!]) \\
\textit{do-receive} &= \lambda\kappa\sigma\pi.\textit{inD}_P\ (\lambda v.\kappa v\sigma\pi) \\
\textit{do-tick} &= \lambda\kappa v\sigma\pi.\textit{inD}_P\ (\textbf{tick}) \bullet \kappa v\sigma\pi \\
\textit{terminate} &= \lambda msg\,v\sigma\pi.\textit{inT}_P\ (\textbf{proc-termin}\ msg)
\end{aligned}
$$

The definition of *merge* is simpler now. *merge* only deals with two (possibly suspended) sequences. We can eliminate the function *filter*, and *merge* becomes directly responsible for scheduling the (simulated) interleaving of the two sequences. The resulting definition, shown in Table 9, is directly derived from the definition in section 2.2. We use a function *randombool* to introduce nondeterminism in the definition of *merge*. Consequently, if both sequences are prefixed by a communication which does not force synchronization, either one may occur first.

The code for the auxiliaries is given in the Appendix.

We are now ready to translate the semantic equations into the language of the transducer generator. *Par abus de language,* we refer to the input to the transducer generator as the transducer. Some key portions of this transducer are shown in Table 10. (The entire transducer may be found in the Appendix).

The transducer is a relatively straightforward bottom-up translator. Syntactically, it is a list of items. The first item in the list specifies the parser, which can be written by hand or generated using an interface to the parser generator *yacc* [5]. The rest of the items in the list are *actions*, one per production. Each action consists of a production name, the production itself (to be used by the type checker and the parser generator), dummy variables, and a body. At each node of the parse tree, the corresponding action

$merge(e_1, e_2) =$
  $casecom\ (first(e_1))$
    $\lambda\, ns_1.\ casecom\ (first(e_2))$
      $\lambda\, ns_2.randombool \rightarrow ns_1 \bullet merge(rest(e_1), e_2),\ ns_2 \bullet merge(e_1, rest(e_2))$
      $\lambda\, re.ns_1 \bullet merge(rest(e_1), e_2)$
      $\lambda\, se.ns_1 \bullet merge(rest(e_1), e_2)$
      $\lambda\, te.ns_1 \bullet merge(rest(e_1), e_2)$
    $\lambda\, re_1.casecom\ (first(e_2))$
      $\lambda\, ns.\ ns \bullet merge(e_1, rest(e_2))$
      $\lambda\, re_2.$ **deadlocked**
      $\lambda\, se.$ **tick** $\bullet\ merge(re_1\ (se), rest(e_2))$
      $\lambda\, te.$ **deadlocked**
    $\lambda\, se_1.casecom\ (first(e_2))$
      $\lambda\, ns.\ ns \bullet merge(e_1, rest(e_2))$
      $\lambda\, re.$ **tick** $\bullet\ merge(rest(e_1), re\,(se_1))$
      $\lambda\, se_2.$ **deadlocked**
      $\lambda\, te.$ **deadlocked**
    $\lambda\, te_1.casecom\ (first(e_2))$
      $\lambda\, ns.\ ns \bullet merge(e_1, rest(e_2))$
      $\lambda\, re.$ **deadlocked**
      $\lambda\, se.$ **deadlocked**
      $\lambda\, te_2.(te_1 \bullet te_2)$

Table 9. Definition of *merge* in the implementation.

is consulted. The values of the subtrees are bound to the dummy variables (like $1, *etc.* in *yacc*), and the body is then evaluated.

The transducer is type-checked by associating a type with each non-terminal symbol. In our case, we declare:

```
(define-type-abbrev Sys
   (-> (seq (list bas) (list bas)) sysans))

(define-type-abbrev Exp
   (-> (seq env cont)
       (-> (seq state) proc-answer)))

(define-type-abbrev Const value)

(define-type-abbrev Ident (literal))
```

Given these declarations, the type checker then tries to confirm that the body of each action has the type of the left-hand side of its production, under the assumption that each dummy variable has the type of its corresponding non-terminal.

With the help of the typechecker the programming exercise was easy. Once the program was typechecked, no further debugging was necessary and the transducer produced the correct answers right away.

6. Conclusions

Starting from the linear history semantics in [1], we gave a new way to extend a language with continuations semantics to multiprocessing. Our method was based on the observation that a process makes irrevocable decisions during its computation. We described the semantics of a process in terms of these decisions. Then we showed a way to combine two processes in order to describe a concurrent system. Eventually we showed how we have to modify our results in order to obtain an implementation of the language. For this implementation we used a transduction facility based on semantic equations. In this way, we demonstrated that we can use a standard continuation semantics framework, add multiprocessing, and implement the result in a straightforward way.

An interesting question is the extension of our method to a system of more than two processes, and to a system where processes are created dynamically. Statically allocated processes seem manageable by using the process names more consistently than we have done in our simple example. Dynamic process creation seems to pose some additional difficulties.

Another theoretical question, not answered in [1] is the following: linear history (or future) semantics uses only a *subset* of the power set of sequences, as not every set of sequences is a legal set of histories (or futures). The set must, for example, have the right closure properties with respect to **not-done-yet** and **tick**. Thus the power set is a representation of some other ordering. We do not know at present just what that ordering is, or what the proper closure properties are. This is a subject for further research. In the meantime, linear future semantics appears to be a useful tool for adding multiprocessing to more realistic languages.

Appendix: Code for Auxiliaries and Transducer

```
(define-checked do-output
```

```
(define semP
   '(parser-for-semP                ; the parser.
      (sys                          ; a production name.
         (Sys 'lparen 'parbegin Exp Exp 'rparen)
         ; the production:
         ; Sys -> (parbegin Exp1 Exp2) .
         (e1 e2)                    ; dummy variables.
         (lambda (w1 w2)            ; the value of the node.
            (merge (((e1 env$init cont$init)
                     (mk-state w1 nil))
                    1)
                   (((e2 env$init cont$init)
                     (mk-state w2 nil))
                    2))))
      (const (Exp Const)
             (c)
             (lambda (r k) (do-tick k c)))
      (ident (Exp Ident)
             (id)
             (lambda (r k)
                (casefn (r id)
                   (lambda (b)
                      (terminate
                         (list "unbound identifier " id)))
                   (lambda (v) (do-tick k v)))))
      (receive
         (Exp 'lparen 'receive 'rparen)
         nil
         (lambda (r k) (do-receive k)))
      (send
         (Exp 'lparen 'send Exp 'rparen)
         (e)
         (lambda (r k) (e r (do-send k))))
      ...))
```

Table 10. Excerpt from transducer for implementation.

```
(-> (seq cont bas state) proc-answer)
```

```
   (lambda (k b s)
      (lambda (p)
         (cons (inP1 (list 'output b))
               (((k (inL b)) s) p)))))

(define-checked do-send
   (-> (seq cont)
       (-> (seq value)
           (-> (seq state) proc-answer)))
   (lambda (k)
      (lambda (v)
         (lambda (s)
            (lambda (p)
               (casefn v
                  (lambda (b)
                     (cons (inP3 b) (((k v) s) p)))
                  (lambda (f)
                     (((terminate
                          (list "can't send fcn"))
                       s)
                      p))))))))

(define-checked do-receive
   (-> (seq cont) (-> (seq state) proc-answer))
   (lambda (k)
      (lambda (s)
         (lambda (p)
            (cons (inP2 (lambda (b) (((k (inL b)) s) p)))
                  nil))))))

(define-checked do-tick
   (-> (seq cont value) (-> (seq state) proc-answer))
   (lambda (k v)
      (lambda (s)
         (lambda (p)
            (cons (inP1 (list 'tick)) (((k v) s) p))))))

(define-checked terminate
   (-> (seq msg) (-> (seq state) proc-answer))
   (lambda (m)
      (lambda (s)
         (lambda (p)
```

```
            (cons (inP4 (list m)) nil)))))
```

The definition of the function **merge** follows exactly the definition given previously:

```
(define-checked merge
   (-> (seq proc-commun-seq proc-commun-seq) sys-answer)
   (lambda (e1 e2)
      (casecom (car e1)
         (lambda (ns1)
            (casecom (car e2)
               (lambda (ns2)
                  (if (randombool)
                      (cons ns1
                         (merge (cdr e1) e2))
                      (cons ns2
                         (merge e1 (cdr e2)))))
               (lambda (re)
                  (cons ns1
                     (merge (cdr e1) e2)))
               (lambda (se)
                  (cons ns1
                     (merge (cdr e1) e2)))
               (lambda (te)
                  (cons ns1
                     (merge (cdr e1) e2)))))
         (lambda (re1)
            (casecom (car e2)
               (lambda (ns)
                  (cons ns
                     (merge e1 (cdr e2))))
               (lambda (re2)
                  (cons (list 'deadlocked) nil))
               (lambda (se)
                  (cons (list 'tick)
                        (merge (re1 se) (cdr e2))))
               (lambda (te)
                  (cons (list 'deadlocked) nil))))
         (lambda (se1)
            (casecom (car e2)
```

```
                (lambda (ns)
                   (cons ns
                      (merge e1 (cdr e2))))
                (lambda (re)
                   (cons (list 'tick)
                         (merge (cdr e1) (re se1))))
                (lambda (se2)
                   (cons (list 'deadlocked) nil))
                (lambda (te)
                   (cons (list 'deadlocked) nil))))
          (lambda (te1)
             (casecom (car e2)
                (lambda (ns)
                   (cons ns
                      (merge e1 (cdr e2))))
                (lambda (re)
                   (cons (list 'deadlocked) nil))
                (lambda (se)
                   (cons (list 'deadlocked) nil))
                (lambda (te2)
                   (cons (list te1 te2) nil)))))))

(define-checked do-read
   (-> (seq cont) (-> (seq state) proc-answer))
   (lambda (k)
      (lambda (s)
         (if (null (lson s))
             ((terminate (list "eof on read")) s)
             ((do-tick k (inL (car (lson s))))
              (mk-state (cdr (lson s)) (rson s)))))))

(define-checked do-print
   (-> (seq cont)
       (-> (seq value)
           (-> (seq state) proc-answer)))
   (lambda (k)
      (lambda (v)
         (lambda (s)
            (casefn v
               (lambda (b)
                  (do-output k
                     b
```

```
                    (mk-state (lson s)
                       (append (rson s)
                          (cons b nil)))))
                (lambda (f)
                   ((terminate
                       (list "can't print fcn"))
                    s)))))))

(define semP
   '(parser-for-semP
      (sys
         (Sys 'lparen 'parbegin Exp Exp 'rparen)
         (e1 e2)
         (lambda (w1 w2)
            (merge (((e1 env$init cont$init)
                     (mk-state w1 nil))
                    1)
                   (((e2 env$init cont$init)
                     (mk-state w2 nil))
                    2))))
      (const (Exp Const)
             (c)
             (lambda (r k) (do-tick k c)))
      (ident (Exp Ident)
             (id)
             (lambda (r k)
                (casefn (r id)
                   (lambda (b)
                      (terminate
                         (list        "unbound identifier " id)))
                   (lambda (v) (do-tick k v)))))
      (error
         (Exp 'lparen 'error Msg 'rparen)
         (msg)
         (lambda (r k)
            (terminate
               (list "error-found: "
                  msg))))
      (if
         (Exp 'lparen 'if Exp 'then Exp 'else Exp 'rparen)
         (e0 e1 e2)
         (lambda (r k)
```

```
      (e0 r
         (lambda (v)
            (if (val-to-bool v)
                (e1 r k)
                (e2 r k))))))
(fn
   (Exp 'lparen 'fn Ident Exp 'rparen)
   (id e)
   (lambda (r k)
      (do-tick k
         (inR
            (lambda (v)
               (lambda (k1)
                  (e (ext r v id)
                     k1)))))))
(rec
   (Exp 'lparen 'rec Ident 'lparen 'fn Ident Exp 'rparen 'rparen)
   (fnid parid body)
   (lambda (r k)
      (do-tick k
         (fix theta
            (inR
               (lambda (v)
                  (lambda (k1)
                     (body
                        (ext (ext r theta fnid) v parid)
                        k1)))))))))
(applic
   (Exp 'lparen Exp Exp 'rparen)
   (e1 e2)
   (lambda (r k)
      (e1 r
         (lambda (v)
            (casefn v
               (lambda (b)
                  (terminate
                     (list
                        (quote
                           "attempt to apply non fcn"))))
               (lambda (f)
                  (e2 r
                     (lambda (a)
```

```
                        ((f a) k)))))))))
(receive
   (Exp 'lparen 'receive 'rparen)
   nil
   (lambda (r k) (do-receive k)))
(send
   (Exp 'lparen 'send Exp 'rparen)
   (e)
   (lambda (r k) (e r (do-send k))))
(read
   (Exp 'lparen 'read 'rparen)
   nil
   (lambda (r k) (do-read k)))
(print
   (Exp 'lparen 'print Exp 'rparen)
   (e)
   (lambda (r k) (e r (do-print k))))
(number
   (Const bas)
   (b)
   (inL b))
(quoted
   (Const bas)
   (b)
   (inL b))
(t
   (Const)
   nil
   (inL bas$t))
(nil
   (Const)
   nil
   (inL bas$nil))
(car
   (Const)
   nil
   (inR
      (lambda (v)
         (lambda (k)
            (casefn v
               (lambda (b)
                  (do-tick k (inL (bas$car b))))
```

```
                (lambda (f)
                   (terminate
                      (list "can't take car of function"))))))))
(cdr
   (Const)
   nil
   (inR
      (lambda (v)
         (lambda (k)
            (casefn v
               (lambda (b)
                  (do-tick k (inL (bas$cdr b))))
               (lambda (f)
                  (terminate
                     (list "can't take cdr of function"))))))))
(atom
   (Const)
   nil
   (inR
      (lambda (v)
         (lambda (k)
            (casefn v
               (lambda (b)
                  (do-tick k (inL (bas$atom b))))
               (lambda (f)
                  (terminate
                     (list "can't take atom of function"))))))))
(cons
   (Const)
   nil
   (inR
      (lambda (v1)
         (lambda (k1)
            (casefn v1
               (lambda (b1)
                  (do-tick k1
                     (inR
                        (lambda (v2)
                           (lambda (k2)
                              (casefn v2
                                 (lambda (b2)
                                    (do-tick k2
```

```
                                                (inL
                                                   (bas$cons b1 b2))))
                                             (lambda (f)
                                                (terminate
                                                   (list "cant-cons-fcn")))))))))
                          (lambda (f)
                             (terminate (list "cant-cons-fcn")))))))
(eq
   (Const)
   nil
   (inR
      (lambda (v1)
         (lambda (k1)
            (casefn v1
               (lambda (b1)
                  (do-tick k1
                     (inR
                        (lambda (v2)
                           (lambda (k2)
                              (casefn v2
                                 (lambda (b2)
                                    (do-tick k2
                                       (bool-to-val
                                          (bas$eq b1 b2))))
                                 (lambda (f)
                                    (terminate
                                       (list "cant-eq-fcn")))))))))
               (lambda (f1)
                  (terminate
                     (list "cant-eq-fcn")))))))))
```

Acknowledgements

Thanks to Christopher T. Haynes and Steven D. Johnson for their detailed comments on the manuscript.

References

[1] Francez, N., Lehmann, D.J. and Pnueli, A. "A Linear History Semantics for Distributed Languages," *21st Annual Symposium on Foundations of Computer Science* (Syracuse, 1980), 143–152.

[2] Gordon, M.J.C. *The Denotational Description of Programming Languages*, Springer, Berlin, 1979.

[3] Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M. "The Scheme 84 Reference Manual" Indiana University Computer Science Department Technical Report No. 153 (March, 1984).

[4] Wand, M. "A Semantic Prototyping System," to appear, *Proc. ACM SIGPLAN '84 Compiler Construction Conference* (1984).

[5] Johnson, S.C. "Yacc: Yet Another Compiler-Compiler" Technical Report CSTR32, Bell Laboratories, Murray Hill, NJ, 1975.