Repairing the PROCESS semantics for parallelism

Maarten Fokkinga
Techn Hogeschool Twente, afd TW
P.O. Box 217 Enschede Netherlands

21 April 1977

Abstract
A separation of the defect in the semantics of the
illustrative programming language in Milner (1973)
is proposed. The semantic equations have (almost)
not been changed; it is the concept of process which
has been slightly altered.

Contents
0 Introduction, References  pag 1
1 Reformulation of the original Semantics  2
2 Alternative concept of Process  3
3 A mistake in the illustrative program?  6
4 Localizing the existence of slave processes  7
5 Elimination of the ORACLE and GENERATOR process  11.
In Folkvmoa (1975) we show the following defect in the semantics as proposed by Milner (1973). Consider the following program:

\texttt{letslave outer be \ldots in letslave inner be \ldots in \ldots outer(inner)\ldots.}

Suppose the main expression \texttt{\ldots outer(inner)\ldots} makes no explicit request of an interrogation of the inner slave. Then the inner slave process completely disappears in the binding of the main expression with the inner slave. But the outer slave-as you see-will receive the inner slave location and subsequently make a request for its interrogation. Thus the illustrative program for adding up numbers from 1 to 100 with parallelism fails.

In this paper we propose a separation. The fact that the semantic equations will (almost) not be changed might be a strong indication that we do not diverge too far from the original intention. It is, in this paper, the concept of Process which has slightly been changed.

I acknowledge the constant requests made by Willem Paul de Roever to fulfill my announcement in Folkvmoa (1975).

The reader is assumed that the reader is familiar with Milner (1973). All terminology, notation, concepts and definitions are taken from his paper.

References:
M Folkvmoa (1975): Comments on a paper by R Milner...,
TW-Memo no 71, THF-Enschede Netherlands.
Reformulation of the original semantics equations

We reformulate the original semantic equations so that there occurs no explicit process denotation. This has the advantage that we may change the semantics - while maintaining the equations literally - by changing the concepts of processes and redefining the operations on processes accordingly.

The semantic equations then become (note: $E : Exp \rightarrow Env \rightarrow P$):

$E[x]r = r[x]$

$E[e(e')]r = E[e]r \ast \lambda v. (E[e]r \ast (v : L | SEND v | L, v | P))!$

$E[e; e']r = E[e]r \ast E[e']r$

$E[\pi x : e]r = \text{QUOTE} (\lambda v. E[e]r \{\text{QUOTE } v / x \})! )\text{\textit{iv v}}$

$E[\text{letrec } x \leftarrow e \text{ in } e']r = E[e']r \quad \text{where } x_r = r \{E[e]r'/x\}$

$E[\text{letrec } x \leftarrow e \text{ in } e']r = E[e]r \ast$

\[\text{NEWLOC } \text{\textit{ilv, SLAVE (vL) (E[e]r \{\text{QUOTE } v / x\}) (v | P)}!}\]

$E[e \ltc e']r = E[e]r \ast E[e']r$

$E[e \para e']r = E[e]r \land E[e']r$

$E[\text{if e then } e' \text{ else } e'']r = E[e]r \ast \text{COND}(E[e']r)(E[e'']r)$

$E[\text{while e do e'}]r = Y \lambda p. E[e]r \ast \text{COND}(E[e']r \ast p) \ast \text{ID}$

$E[e \\text{ renew e'}]r = E[e']r \ast \lambda v. (E[e]r \ast v | P)!$

$E[(e)]r = E[e]r$

$E[]r = \text{QUOTE !}$

The only changes are caused by SEND and NEWLOC:

$\text{SEND (e L } \rightarrow P) = \lambda x. \lambda v. (<x, v, ID>)$

$\text{NEWLOC (e (v } \rightarrow P) } \rightarrow P) = \lambda f. \lambda v. (<v, !, v, f \ \text{Lv, v}>)$

SLAVE is merely synonymous for BIND: it is not too suggestive.

Finally, it appears that the renewal of a parallel composition is forced to be L, whereas we wish it should be the parallel composition of the renewals of the original processes. So we define:
2 Alternative concept of Process

A Process (the entity denoted by a piece of program text) is an object which may be interrogated by a value and then either requests or indicates an interrogation of another process by some value and in both cases potentially creates (and destroys?) other slave processes. All the processes together form the state of the system.

Formally:

\[ P = V \to C \times L \times V \times P \quad \{ \text{Processes} \} \]
\[ S = L \to P \quad \{ \text{States: associations of Processes to Locations} \} \]
\[ C = \{ \text{Changes of state} \} \]

A possible implementation of \( C \) might be:

\( C = L \to (\text{Augm} + \text{Del} + \text{Nothing}) \)

\( \text{Augm} = P \)

\( \text{Del, Nothing are immaterial; they only serve the purpose of ease distinction.} \)

Thus \( C \in C \) indicates at which locations the state should be updated: augmented by a given process or just be destroyed.

For ease of description we choose another implementation of \( C \):

\( C = S \to S \).
It might well be argued that this space is much too large: it will be easily provable that the former implementation is sufficient. The accumulation of changes to be made, however, is now easy explicable by functional composition.

All the original operations on processes have their obvious analogous forms. Here we list them.

\[
\text{ID}(e P) = \lambda v. \langle \text{id}, v, v, \perp \rangle
\]

\[
\text{QUOTE}(e \overrightarrow{K} \rightarrow \overrightarrow{P}) = \lambda v. \overrightarrow{K}(\text{id} v)
\]

\[
\text{SEND}(e \overrightarrow{L} \rightarrow \overrightarrow{P}) = \lambda \alpha . \lambda v. \langle \text{id}, \alpha, v, \perp, \text{id} \rangle
\]

\[
\text{NEWLOC}(e \overrightarrow{N} \rightarrow \overrightarrow{P}) = \lambda f . \lambda v. \langle \text{id}, v', \perp, \lambda w. f \overrightarrow{w} \rangle
\]

and

\[
\text{COND}(e \rightarrow P \rightarrow P) = \lambda p, q . \langle \lambda v . v/I \rightarrow P !, q ! \rangle
\]

\[
\text{EXTEND}(e \rightarrow P \rightarrow P) = \lambda p, q . \lambda w . \langle \text{id}, w, !, \lambda v . w/I \rightarrow P !, q ! \rangle
\]

\[
\text{?}(e \rightarrow P \rightarrow P) = \lambda p, q . \lambda v . \langle \text{id}, w, !, \lambda v . w/I \rightarrow P !, q ! \rangle
\]

\[
\text{!}(e \rightarrow P \rightarrow P) = \lambda p, q . \lambda v . \langle \text{id}, w, !, \lambda v . w/I \rightarrow P !, q ! \rangle
\]

\[
\text{SLAVE}(e \overrightarrow{L} \rightarrow \overrightarrow{P} \rightarrow \overrightarrow{P}) = \lambda \alpha, v. \text{firstly } (\alpha/\overrightarrow{q}) P
\]

\[
\text{firstly } (e \overrightarrow{N} \rightarrow \overrightarrow{P} \rightarrow \overrightarrow{P}) = \lambda t, p . \lambda v . \langle t', v', t, \overrightarrow{p} \rangle
\]

\[
\text{firstly } (e \overrightarrow{S} \rightarrow \overrightarrow{P} \rightarrow \overrightarrow{P}) = \lambda t, p . \lambda v . \langle t', v', t, \overrightarrow{p} \rangle
\]

Note: \{\alpha/\overrightarrow{q}\} denotes the function \(e \overrightarrow{S} \rightarrow \overrightarrow{S}\).
Finally, the purely extensional meaning of expressions is defined as follows:

\[ \text{MNG}(\text{Exp} \rightarrow \text{Tb}) = \text{IE. Run (E|I|I|b)} \text{ So } \]

\[ \text{Tb} = V \rightarrow V \times Tb \{ \text{Transducer behaviours} \} \]

\[ \text{Run} (\text{E P} \rightarrow \text{S} \rightarrow \text{Tb}) = \lambda p, s. \]

\[ \text{let } <u', s'> = \text{RUN } s' \{ p \} \text{ in } <u', \text{Run } s' ' u > \]

\{ The main process \( p \) is located at \( u \) in the state \( s \) and \}

\{ the state is triggered at the process \( u \) by means of \( \text{RUN} \) \}

\[ \text{RUN} (\text{E S} \rightarrow \text{L} \rightarrow \text{X} \text{ where } \text{X} = V \rightarrow V \times X) = \lambda s, x. \]

\{ The slave process located at \( s \) is executed in state \( s \) until \}

\{ it yields a result ; slave processes, for which a request for \}

\{ an interrogation is made, will be executed firstly in the \}

\{ same way \}

\[ \lambda u. \text{let } <t', l', u', p'> = s \times u \text{ and } s' = (t', s) \{ \langle p' \rangle \}

\{ \text{where } l' = \text{RUN } s' \text{ in } \text{RUN } s'' \times u'' \].

Of course, \( e_0 \) is the standard environment and might be

"empty", and so is the initial state and must contain

a generator process \( V \) and an oracle process \( U \).

(The operation \( \text{RUN} \) is precisely the kind of "binding" as pro-

posed in Hoogkinga.(1975))

This completes the formal semantics. We hope that they

are the intended one. In the next section, however, we

show that according to these semantics the program for

adding up the numbers from 1 to 100 with parallelism

is not correct! So either the semantics do not reflect

the original intentions or the program writer has made

a mistake. (We hope the latter to be true!)}
A mistake in the illustrative program?

Even if the action of delivering a process is considered atomic, the delivered process itself is not necessarily treated atomic. So in the scope of "let slave Inc be reset \( \pi x . \pi y . x := x + y \)" the actions from the triggering of Inc up to the return of the value (the process denoted by \( \pi y . x := x + y \)) will be treated as an atomic action (there is however only one single step involved: \texttt{QUOTE}). But the application of the value (the process) to some argument might be interleaved with other processes.

The following declarations might work well:

\[
\text{let slave Inc be reset } \pi x . (\text{let slave incr be } \pi y . x := x + y \text{ in incr}), \\
\text{let tree Inc be } \pi x . (\ldots).
\]

In general, when \( e \) denotes a \texttt{QUOTE} process, then

\[
\text{let slave } s \text{ be reset } \pi x . e \equiv \text{let tree } s \text{ be } \pi x . e.
\]

In both cases, the actions —if any— induced by the value delivered by the \texttt{QUOTE} process denoted by \( e \) need not necessarily treated atomic. Examples of such \( e \) are: \( \pi y . e! \) and \( \text{let slave incr be } \pi y , \ldots \text{ in incr} \).
It is clear from the semantic equations that a slave process, once it has been created, exists forever: no deletion has been specified anywhere. The following program scheme shows that one might think it to be necessary.

\[ \text{let slave s be (let slave own be cell(a) in next r x...)} \text{ in expr.} \]

Indeed, 'own' is a "private global variable" of the slave process and it should still exist whenever \( s \) is called. Thus the life time of 'own' is not restricted to the evaluation of the let slave expression in which it has been declared. This may hold for every slave in the program.

Suppose we pose some restrictions on the language so that the above program is not allowed. Then we like to define the semantics so that a slave process is annihilated as soon as "control leaves the subexpression" in which the slave has been declared locally. However control may stay forever in the subexpression (cfr while true do write(read(c))). So the deletion of the slave process from the state cannot be specified inside the semantic equation of the let slave expression. It seems possible to redefine \( E \) such that the deletion is taken care for in the remaining semantic equations where the processes of subexpressions are serially composed. We will not pursue the subject because below we present a much nicer and more general solution for the problem of localizing the existence of slave processes.

Even if there is no restriction on the language one might wish to reflect in the semantics that a slave process
only can be accessed from within its master process [the
process denoted by the main expression of the let-expressions con-
struction] and consequently only need to exist—having
the right value—when its master process has been inter-
rogated but not yet produced a result. Thus we let
each process delete its slave processes from the state
when it produces a result and restore the slave
processes—with the right value—at the next interro-
gation. As a consequence, when a process is serially
composed with another one, the deletion of slave
processes is performed implicitly and so we have an
elegant solution for the problem posed § above as well.

Because a renewal process has to restore what has been
deleted previously from the state, its specification is ob-
viously dependent on the state. Thus we are led to change
from \( P = V \rightarrow (S \rightarrow S) \times L \times V \times P \) to \( B = V \rightarrow L \times V \times (S \rightarrow S \times P) \). Below
we will firstly redefine all operations on processes such that
the semantics of the language remains the same. Then
we will adapt the SLAVE operation to take care of the
above described ideas.

\[
P = V \rightarrow L \times V \times (S \rightarrow S \times P)
\]

The redefinitions of the operations is rather straight-
forward—only EXTEND requires some attention.

\[
\text{ID}(B) = \lambda v. < v, v, \text{AS}(s, l)>
\]

\[
\text{QUOTE}(E \rightarrow V) = \lambda v. \text{K}(\text{ID}(v))
\]

\[
\text{SEND}(E \rightarrow L) = \lambda x. \lambda v. < x, v, \text{ID}>
\]

\[
\text{NEWLOC}(E \rightarrow V \rightarrow P) = \lambda f. \lambda u. < v, !, \text{AS}(s, \lambda v. f \circ v)>
\]

and

\[
\text{COND}(E \rightarrow P \rightarrow P) = \lambda p, q. (\lambda v. \text{IF}\_T \rightarrow p \!, q \!)
\]

\[
\text{EXTEND}(E \rightarrow (S \rightarrow P) \rightarrow P) = \lambda p, f.
\]

\[
\text{convention}: \text{if t.e.s} \rightarrow S \times P \quad \text{then} \quad \text{ls}(E \rightarrow S) = \text{AS}(t, s), \quad \text{and}
\]

(need line)
\[ t_2 (e_s \rightarrow p) = \lambda s. (t_s)_2 \]

\[ \lambda v. \text{let } \langle e, v, t \rangle = pv \in l = 2 \Rightarrow \text{firstly } e_1 (f, t_2) \nu, \langle e, v, \lambda s. (t_4 s \rightarrow \text{extend } (t_2 s) f) \rangle \]

\[ \ast (e_p \rightarrow p \rightarrow p) = \lambda p, q. \text{extend } p (k q) \]

\[ \gamma (e_p \rightarrow p \rightarrow p) = \lambda p, q. \lambda v. \langle w, 1, \lambda s. (s, \lambda v'. v' | T \rangle C p v, q v \rangle \]

\[ \eta (e_p \rightarrow p \rightarrow p) = \lambda p, q. \quad p \ast \lambda v. \langle w, 1, \lambda s. (s, q) \rangle \]

\[ \eta (e_p \rightarrow p \rightarrow p) = \lambda p, q. \quad \lambda v. \langle w, 1, \lambda s. (s, q) \rangle \]

\[ \text{let } w = v \backslash w \text{ in } \lambda s. (s, \]

\[ \lambda w. \text{ if } w | T \]

\[ \text{then let } \langle e, v, t \rangle = pw, \]

\[ \text{in } l = 2 \Rightarrow \text{extend } (f, t_2 q) \]

\[ (w' \lambda v'. \langle e, v, \lambda s. (s, t_2 s) \rangle \backslash (t_2 s)) \]

\[ \text{else similarly } \}

\[ \text{SLAVE } (e_p \rightarrow p \rightarrow p) = \lambda x, p, q. \text{firstly } (f / q) \]

\[ \text{let } \langle e, v, t \rangle = pv \in \langle e, v, \lambda s. (f / q) o_1 s, t_2 s \rangle \]

\[ \text{firstly } (e (s \rightarrow p) \rightarrow p) = \lambda b, p. \]

\[ \lambda v. \text{let } \langle e', v', t' \rangle = pv \in \langle e', v', \lambda s. (t_4 o_1 t s, t_2 s) \rangle \]

Finally, RUN needs to be redefined.

\[ \text{RUN } (e s \rightarrow l \rightarrow x \text{ where } x = v \rightarrow v \times x) = \lambda s, x. \]

\[ \lambda v. \text{let } \langle e, v, t \rangle = s \circ v \text{ and } s' = (f / t' s) \circ t_1 s \]

\[ \text{in } l = 2 \Rightarrow \langle v', s' \rangle, \text{let } \langle v', s'' \rangle = \text{RUN } s' \circ e' \circ v' \text{ in } \text{RUN } s'' \circ v''. \]

It should be easy to prove the following

Theorem. "The semantics have not been changed".

Actually, a much stronger property is provable:

Theorem. "RUN has not been changed".

Thus slave processes still exist forever.

Now we redefine the SLAVE operation so that slaves do not exist in between two activations of their master.
process. To this end we define
\[ \text{loc} (\ell \to P \to P) = \lambda x, p. \]
\[
\{ p \text{ is adapted in that } \ell \text{ localizes (deletes & restores) the slave at } x \}
\]
\[ \nu x. \lambda e_n < e, v, t > = p v \]
\[ \text{in } e = n \Rightarrow < e, v, x s. \langle \text{del } x \rangle (e_s), \text{loc } x (\text{firstly } (x/s x) (e_s)) \rangle >, \]
\[ < e, v, x s. \langle t s, \text{loc } x (t_s) >, \]
\[ \text{del } (\ell \to B \to B) = \lambda x. \{ x/\alpha \text{ immaterial...} \}. \]

Indeed, the definition is now straightforward:
\[
\text{SLAVE} (\ell L \to P \to P \to P) = \lambda x, p, q. \text{loc } x (\text{firstly } (x/q) p) \]

Again it should be easy to prove the following

Theorem. "The semantics have not been changed".

But now the slave processes only exist when necessary?

Of course, the above results can also be obtained with
the following notion of process: \( P = S x V \to L x V x P x S \). This
was indeed one of my previous attempts; but contrary to
the above approach, the given state was interpreted as
exactly the state in which the process step should be
carried out. Hence, the augmenting state by a
given slave process was rather unexpected: the state
transformation should also reflect the substitution of
the process by its remainder process. It took me several
weeks to discover an erroneous augment definition made
in the first day of this development.

The fact that the original semantic equations remain
unaltered is a strong indication that we have not
diverged too far from the original intention.
In my point of view the oracle and generator process should be of no concern for the programmer. The semantics however forces the program writer to consider them as processes potentially interacting with user defined processes. Indeed, without the specification of both processes it is not possible to prove the absence of interaction.

The following alterations in the semantic definitions make the absence of interaction immediately clear.

1. Change from \( P = V \rightarrow (S \rightarrow S) \times L \times V \times P \) or \( P = V \rightarrow L \times V \times (S \rightarrow S) \times P \)
   to \( P = V \times S \rightarrow L \times V \times P \times S \)

2. Redefine all operations on processes in the obvious way.

3. Replace in the semantic equations everywhere \( '\lambda v' \) and \( '\lambda u' \) by \( '\lambda v,s' \) and \( '\lambda u,s' \); and \( '!' \) by \( 's!' \).

Now make the following semantic changes:

\[ S = (L \rightarrow P) \times L \times S, \]
\[ L = \text{integer} \{ \text{simple implementation of locations} \}, \]
\[ S = T \times S \{ \text{a sequence of truth values} \}. \]

Thus a state consists of the association \( L \rightarrow P \), the last distributed location and the remainder oracle.

\[ \text{NEWLOC} (\epsilon (V \rightarrow P) \rightarrow P) = \lambda f. \]
\[ \lambda u,s. \text{let } p = f u \text{ and } v = s_2 + 1 \text{ and } s' = \langle s_1, s_2 + 1, s_3 \rangle \]
\[ \text{in } p \cdot v \cdot s'. \]

Finally, replace the two tuples \( <s,v,!, \tau,s',v'> \rightarrow \) by

\[ \text{let } s' = <s_1, s_2, (s_3)_2 \rangle \text{ and } v' = (s_3)_4 \text{ in } \ldots \rightarrow \]