Repairing the PROCESS semantics for parallellism

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

21 April 1977

Abstract
A reparation 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

In Fokkinga (1975) we show the following defect in the semantics as proposed by Milner (1973). Consider the following program:

<u>letslave</u> outer <u>be</u> ... <u>in</u> <u>letslave</u> inner <u>be</u> ... <u>in</u> ---outer (inner)---.

Suppose the main expression ---outer (inner)--- 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 recieve 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 parallellism fails.

In this paper we propose a reparation. The fact that the semantic equations ~~do not~~ 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 fullfil my announcement in Fokkinga (1975).

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

<u>References</u>:

R Milner (1973): An Approach to the Semantics of Parallel Programs, Techn Memo, Univ of Edinburgh.

M Fokkinga (1975): Comments on a paper by R Milner..., TW-memo no 71, THT - Enschede Netherlands.

1. Reformulation of the original semantics ~~equations~~

We reformulate the original semantics 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: $E \in Exp \to Env \to P$):

$$E[\![x]\!]r = r[\![x]\!]$$

$$E[\![e(e')]\!]r = E[\![e]\!]r * \lambda v.\,(E[\![e']\!]r * (v{:}L \supset \mathrm{SEND}\ v|L\ ,\ v|P))!$$

$$E[\![e;e']\!]r = E[\![e]\!]r * E[\![e']\!]r$$

$$E[\![\pi x.e]\!]r = \mathrm{QUOTE}\,(\lambda v.\ E[\![e]\!]r\{(\mathrm{QUOTE}\ v)/x\}\ !\ )\mathrm{in}V$$

$$E[\![\underline{\text{letrec}}\ x\ \underline{\text{be}}\ e\ \underline{\text{in}}\ e']\!]r = E[\![e']\!]r' \qquad \text{whererec } r' = r\{(E[\![e]\!]r')/x\}$$

$$E[\![\underline{\text{letslave}}\ x\ \underline{\text{be}}\ e\ \underline{\text{in}}\ e']\!]r = E[\![e]\!]r *$$
$$\mathrm{NEWLOC}\ \lambda u\lambda v.\mathrm{SLAVE}\ (v|L)\ (E[\![e']\!]r\{(\mathrm{QUOTE}\ v)/x\})\ (u|P)\ !$$

$$E[\![e\ \underline{\text{or}}\ e']\!]r = E[\![e]\!]r\ ?\ E[\![e']\!]r$$

$$E[\![e\ \underline{\text{par}}\ e']\!]r = E[\![e]\!]r\ /\!/\ E[\![e']\!]r$$

$$E[\![\underline{\text{if}}\ e\ \underline{\text{then}}\ e'\ \underline{\text{else}}\ e'']\!]r = E[\![e]\!]r * \mathrm{COND}\,(E[\![e']\!]r)\,(E[\![e'']\!]r)$$

$$E[\![\underline{\text{while}}\ e\ \underline{\text{do}}\ e']\!]r = Y\lambda p.\ E[\![e]\!]r * \mathrm{COND}\,(E[\![e']\!]r * p)\ \mathrm{ID}$$

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

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

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

The only changes are caused by SEND and NEWLOC:

$$\mathrm{SEND}\,(\in L \to P) = \lambda\alpha.\ \lambda v.\ \langle \alpha, v, \mathrm{ID}\rangle$$

$$\mathrm{NEWLOC}\,(\in (V \to P) \to P) = \lambda f.\ \lambda u.\langle \nu, !, \lambda v.\ f\ v\rangle$$

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 $\perp$, whereas we wish it should be the parallel composition of the renewals of the original processes. So we define:

$$\|(\in P \to P \to P) = \lambda p, q. \quad \lambda v. < \omega, !, \quad \underline{\text{let}}\ w = v|W\ \underline{\text{in}}$$

$$\lambda v_\omega.\ \underline{\text{if}}\ v_\omega|T$$

$$\underline{\text{then}}\ \underline{\text{let}}\ \langle l', v', p' \rangle = p\,W_1$$

$$\underline{\text{in}}\ \ l' = \tau \supset \text{EXTEND}\ q\ (\lambda q'\ \lambda v''. \langle \tau, (v', v'')\text{in}V, p' \| q' \rangle)\ w_2$$

$$,\ \langle l', v', \lambda v''. (p' \| q)\ (v'', w_2)\text{in}V \rangle$$

$$\underline{\text{else}}\ \text{---- similarly ----} \qquad \rangle.$$

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 produces an answer or ~~indicates~~ requests 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:

$P = V \to C \times L \times V \times P$ { Processes },

$S = L \to P$ { States: associations of Processes to Locations },

$C =$ { Changes of state }.

A possible implementation of C might be:

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

$\text{Augm} = P$,

Del, Nothing are immaterial: they only serve the purpose of case 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 expressible by functional composition.

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

$ID\ (\in P) = \lambda v.\ \langle id, \iota, v, \perp \rangle$

$QUOTE\ (\in V \to P) = \lambda v.\ K\ (ID\ v)$

$SEND\ (\in L \to P) = \lambda \alpha.\ \lambda v.\ \langle id, \alpha, v, ID \rangle$

$NEWLOC\ (\in (V \to P) \to P) = \lambda f.\ \lambda u.\ \langle id, \cancel{\iota}\ !, \lambda v.\ f\ u\ v \rangle$

and

$COND\ (\in P \to P \to P) = \lambda p, q.\ (\lambda v.\ v|T \supset p\,!, q\,!)$

$EXTEND\ (\in P \to (P \to P) \to P) = \lambda p, f.$

$\quad \lambda v.\ \underline{let} \langle t', l', v', p' \rangle = p\,v\ \underline{in}\ l' = \iota \supset firstly\ t'\ (f\,p')\ v',\ \langle t', l', v', EXTEND\ p' f \rangle$

$*\ (\in P \to P \to P) = \lambda p, q.\ EXTEND\ p\,(K\,q)$

$?\ (\in P \to P \to P) = \lambda p, q.\ \lambda v.\ \langle id, \omega, !, (\lambda v'.\ v'|T \supset p\,v, q\,v) \rangle$

$:\ (\in P \to P \to P) = \lambda p, q.\ p * \lambda v.\ \langle id, \iota, v, q \rangle$

$\|\ (\in P \to P \to P) = \lambda p, q.\ \lambda v.\ \langle id, \omega, !,\ \underline{let}\ w = v|W\ \underline{in}$

$\quad \lambda v_w.\ \underline{if}\ v_w|T$

$\qquad \underline{then}\ \underline{let}\ \langle t', l', v', p' \rangle = p\ w_1$

$\qquad\quad \underline{in}\ l' = \iota \supset (EXTEND\ (firstly\ t'\ q)\ (\lambda q', v''.\ \langle id, \iota, (v', w'')\,in\,V, p' /\!/ q' \rangle)$

$\qquad\qquad)\ w_2$

$\qquad\qquad ,\ \langle t', l', v', \lambda v''.\ (p' /\!/ q)\ \langle v'', w_2 \rangle in V \rangle$

$\qquad \underline{else}$ ---- similarly ---- $\rangle$

$SLAVE\ (\in L \to P \to P \to P) = \lambda \alpha, p, q.\ firstly\ (\{\alpha/q\})\ p$

~~$\lambda v.\ \underline{let} \langle t', l', v', p' \rangle = p\,v\ \underline{in}\ \langle \{\alpha/q\} \circ t', l', v', p' \rangle$~~

{Note: $\{\alpha/q\}$ denotes the function $(\in S \to S)\ \lambda s.\lambda l.\ l = \alpha \supset q, s\,l$ }

$firstly\ (\in (S \to S) \to P \to P) = \lambda t, p.$

$\quad \lambda v.\ \underline{let} \langle t', l', v', p' \rangle = p\,v\ \underline{in}\ \langle t' \circ t, l', v', p' \rangle$

{performs the ~~first~~ given transformation first of all}

Finally, the purely extensional meaning of expressions is defined as follows:

$MNG(\in Exp \to Tb) = \lambda e.\ Run\ (E[\![e]\!]r_0)\ s_0$,

$Tb = V \to V \times Tb$ { Transducer behaviours}.

$Run\ (\in P \to S \to Tb) = \lambda p, s.$

$\lambda v.$ $\underline{let}$ $\langle v', s' \rangle = RUN\ \ s\{r/p\}\ \ r\ \ v\ \ \underline{in}\ \ \langle v', Run\ \ s'\ r \rangle$

{the main process p is located at r in the state s and then the state is triggered at the process r by means of RUN}.

$RUN\ (\in S \to L \to X\ \ \text{where}\ X = V \to V \times X) = \lambda s, \alpha.$

{the slave process located at α 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}

$\lambda v.$ $\underline{let}$ $\langle t', l', v', p' \rangle = s\ \alpha\ v$ $\underline{and}$ $s' = (t'\ s)\{\alpha/p'\}$

$\underline{in}$ $l' = r \supset \langle v', s' \rangle$, $\underline{let}$ $\langle v'', s'' \rangle = RUN\ s'\ l'\ v'$ $\underline{in}$ $RUN\ s''\ \alpha\ v''$.

Of course, e_0 is the standard environment and might be "empty", and s_0 is the initial state and must contain a generator process at ν and an oracle process at ω. (The operation RUN is precisely the kind of "binding" as proposed in Fokkinga (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 parallellism 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!)

3 A mistake in the illustrative program?

Even if the action of delivering a process is considered atomic, the delivered process itself is not neccessarily treated atomic. So in the scope of "__letslave__ 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: QUOTE). But the application of the value (the process) to some argument might be interleaved with other processes

The following declarations might work well:

__letslave__ Inc __be__ __reset__ $\pi x .$ (__letslave__ incr __be__ $\pi y . \ x := x+y$ __in__ incr),

__letree__ Inc __be__ $\pi x .$ (").

In general, when e denotes a QUOTE process, then

__letslave__ s __be__ __reset__ $\pi x . e$ $\approx$ __letree__ s __be__ $\pi x . e$.

In both cases, the actions – if any – induced by the value delivered by the QUOTE process denoted by e need not neccessarily treated atomic. Examples of such e are: $\pi y . e'$ and (__letslave__ incr __be__ ~~πy~~ .... __in__ incr).

4 Localizing the existence of slave processes

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 neccessary.

__letslave__ s __be__ (__letslave__ own __be__ cell(a) __in__ __reset__ πx.----) __in__ expr.

Indeed, 'own' is a "private global variable" of the slave process s and it should still exist whenever s is called. Thus the life time of 'own' is not restricted to the evaluation of the letslave 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~~ such 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())). So the deletion of the slave process from the state can not be specified inside the semantic equation of the letslave expression. It seems possible to redefine P such that the deletion is taken care for in the remaining semantic equations where ~~such~~ 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 letslave construction} and consequently only need to exist – having the right value – when its master process has been interrogated 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 interrogation. As a consequence, when a process is serially composed with another one, the deletion of slave processes is performed implictly and so we have a 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 obviously dependent on the state. Thus we are led to change from $P = V \to (S \to S) \times L \times V \times P$ to $P = V \to L \times V \times (S \to S \times P)$. 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.

$P = V \to L \times V \times (S \to S \times P)$.

The redefinitions of the operations is rather straightforward – only EXTEND requires some attention.

$ID\ (\in P) = \lambda v.\ \langle \iota, v, \lambda s.(s, \bot) \rangle$

$QUOTE\ (\in V \to P) = \lambda v.\ K\,(ID\ v)$

$SEND\ (\in L \to P) = \lambda \alpha.\ \lambda v.\ \langle \alpha, v, ID \rangle$

$NEWLOC\,(\in (V \to P) \to P) = \lambda f.\ \lambda u.\langle \nu, !, \lambda s.\ (s, \lambda v. f\,u\,v) \rangle$

and

$COND\ (\in P \to P \to P) = \lambda p, q.\ (\lambda v.\ v|T \supset p!\ ,\ q!)$

$EXTEND\ (\in P \to ((S \to P) \to P) \to P) = \lambda p, f.$

{convention: if $t \in S \to S \times P$ then $t_1 (\in S \to S) = \lambda s.(t\,s)_1$ and

(next line)

$t_2 (\in S \to P) = \lambda s.\ (t\,s)_2 \quad \}$

$\lambda v'.$ let $\langle \ell, v, t \rangle = p\,v'$

in $\ell = \tau \supset$ firstly $t_1\,(f\,t_2)\,v$, $\langle \ell, v, \lambda s.(t_1 s, \text{EXTEND}\,(t_2 s)\,f)\rangle$

$*(\in P \to P \to P) = \lambda p, q.\ \text{EXTEND}\ p\ (K\,q)$

$?(\in P \to P \to P) = \lambda p, q.\ \lambda v.\langle \omega, !, \lambda s.(s, \lambda v'.\ v'|T \supset p\,v, q\,v)\rangle$

$:(\in P \to P \to P) = \lambda p, q.\ p * \lambda v.\langle \tau, v, \lambda s.(s, q)\rangle$

$//(\in P \to P \to P) = \lambda p, q.\ \lambda v.\ \langle \omega, !,$ let $w = v|W$ in $\lambda s.(s,$

$\lambda v_w.$ if $v_w|T$

then let $\langle \ell, v, t\rangle = p\,w_1$

in $\ell = \tau \supset$ EXTEND (firstly t_1 q)

$(\lambda t'\,\lambda v'.\langle \tau, \langle v, v'\rangle \text{in} V, \lambda s.(s, (t_2 s)//(t'_2 s))\rangle)$

w_2

, $\langle \ell, v, \lambda s.(t_1 s, (t_2 s)//q)\rangle$

else ---- similarly ----- $)\rangle$

SLAVE $(\in L \to P \to P \to P) = \lambda \alpha, p, q.$ firstly $(\{\alpha/q\})$ p

~~$\lambda v'.$ let $\langle \ell, v, t\rangle = p\,v'$ in $\langle \ell, v, \lambda s.((\{\alpha/q\} \circ t_1)\,s, t_2 s)\rangle$~~

firstly $(\in (S \to S) \to P \to P) = \lambda t, p.$

$\lambda v.$ let $\langle \ell', v', t'\rangle = p\,v$ in $\langle \ell', v', \lambda s.((t'_1 \circ t)\,s, t'_2 s)\rangle$

Finally, RUN needs to be redefined.

RUN $(\in S \to L \to X$ where $X = V \to V \times X) = \lambda s, \alpha.$

$\lambda v.$ let $\langle \ell', v', t'\rangle = s\,\alpha\,v$ and $s' = (\{\alpha / t'_2 s\} \circ t'_1)\,s$

in $\ell' = \tau \supset \langle v', s'\rangle$, let $\langle v'', s''\rangle = \text{RUN}\ s'\ \ell'\ v'$ in RUN $s''\ \alpha\ v''$.

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

$loc\ (\in L \to P \to P) = \lambda\alpha, p.$

{p is adapted in that it localizes (deletes & restores) the slave at α}

$\lambda v'.$ let $\langle \ell, v, t \rangle = p\,v'$

in $\ell = \iota \supset \langle \ell, v, \lambda s. \langle (del\ \alpha)(t_1 s), loc\ \alpha\ (firstly\ (\{\alpha / s\alpha\})(t_2 s))\rangle\rangle$

$, \langle \ell, v, \lambda s. \langle t_1 s, loc\ \alpha\ (t_2 s)\rangle\rangle,$

$del\ (\in L \to S \to S) = \lambda\alpha.$ {α/... immaterial...}.

Indeed, the definition is now straightforward:

$SLAVE\ (\in L \to P \to P \to P) = \lambda\alpha, p, q.\ \ loc\ \alpha\ (firstly\ (\{\alpha/q\})\ p)$

Again it should be easy to prove the following

Theorem. "The semantics have not been changed".

But now the slave processes only exist when neccessary!

Of course, the above results can also be obtained with the following notion of process: $P = S \times V \to L \times V \times P \times S$. 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, ~~where~~ the augment~~ing the~~ation of a 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 devellopment.

The fact that the original semantic equations remain unaltered is a strong indication that we have not diverged too far from the original intention.

5 Elimination of the ORACLE and GENERATOR process

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 absense of interaction.

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

1. Change from $P = V \to (S \to S) \times L \times V \times P$ or $P = V \to L \times V \times (S \to S \times P)$ to $P = V \times S \to L \times V \times P \times S$
2. Redefine all operations on processes in the obvious way. Don't forget RUN.
3. Replace in the semantic equations everywhere 'λv' and 'λu' by '$\lambda v, s$' and '$\lambda u, s$'; and '!' by 's !'.

Now make the following semantic changes:

$S = (L \to P) \times L \times \Omega$,

$L =$ integer {simple implementation of locations},

$\Omega = T \times \Omega$ {a sequence of truth values}.

Thus a state consists of the association $L \to P$, the last distributed location and the remainder oracle.

$\mathrm{NEWLOC}\,(\in (V \to P) \to P) = \lambda f.$

$\quad \lambda u, s.$ __let__ $p = f\,u$ __and__ $v = s_2 + 1$ __and__ $s' = \langle s_1, s_2 + 1, s_3 \rangle$

$\qquad$ __in__ $p\ v\ s'$.

Finally, replace the two tuples $\langle s, w, !, \lambda s', v'. \text{---} \rangle$ by

__let__ $s' = \langle s_1, s_2, (s_3)_2 \rangle$ __and__ $v' = (s_3)_1$ __in__ ---.

—— // ——