TECHNISCHE HOGESCHOOL TWENTE

MEMORANDUM NR. 71.

COMMENTS ON A PAPER BY R. MILNER CONCERNING THE SEMANTICS OF PARALLELLISM.

DOOR

M. FOKKINGA

MAART 1975.

ONDERAFDELING DER TOEGEPASTE WISKUNDE

Abstract

In part 1 we present a representation of processes in the form of drawings. The representation is less precise as the λ- notation of Milner, but may sometimes provide better insight.
In part 2 we indicate a serious designmistake for the programming language proposed by Milner. We propose another semantics for the same syntax, but we are not satisfied with it.
Throughout this paper, the paper referred to is R Milners "an approach to the semantics of parallel programs" [1]. We assume the reader to be familiar with it.

Contents | Page

Abstract.
Part 1 Another representation of processes 1
.1 Processes 1
.2 Operations on processes 3
.3 Some examples. 6
Part 2 Defect of the semantics 11
.1 The effect of binding, the defect of the semantics. 11
.2 More comments on the program. 14
.3 An attempt to repair the semantics. 16
References. 18

PART 1 Another representation of processes.

Although in general less precise, the representation sketched below may give better insight in some cases. It gives a good understanding of the behaviour of processes during "execution". In fact the only way of binding processes without the help of a λ-calculus program on a computer is by the method proposed here.

.1 Processes

The space of processes P is $P = V \to L \times V \times P$, where V is the space of values, and L is the space of locations. In general a process can be denoted by, e.g.

$p = \lambda v_1 . < l_1 v_1, f_1 v_1, p_2 >$ where

$p_2 = \lambda v_2 . < l_2 v_2, f_2 v_2, p_3 >$ where

$p_3 = \lambda v_3 . < l_3 v_3, f_3 v_3, p_4 >$ where

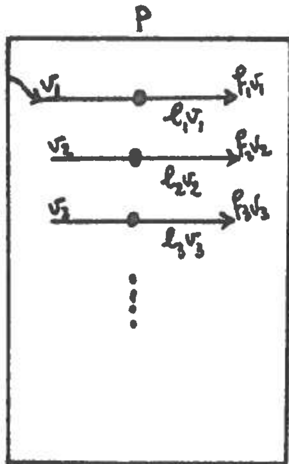
$p_4 =$
$\vdots$

which, when written on one line, is the same as

$\lambda v_1 . < l_1 v_1, f_1 v_1, \lambda v_2 . < l_2 v_2, f_2 v_2, \lambda v_3 . < l_3 v_3, f_3 v_3, \ldots >>>$.

Note that l_i, f_i may depend on v_j for $j < i$.

The new representation of this sample process is

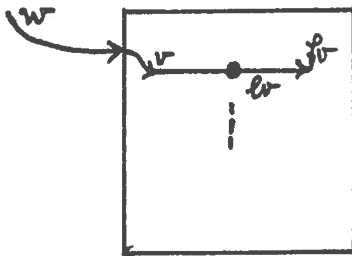


where, as noted above, each l_i and f_i may depend on the v_j for $j < i$; moreover, each v_j is to be read as a bound variable with all beneath it as its scope.

When no confusion results we omit the subscripts of the variables; in particular when the l_i and f_i do not depend on any v_j for $j < i$.

Note that v ——•—— fv → (lv below the dot) corresponds to $\lambda v. < lv,\ fv, \ldots\ >$.

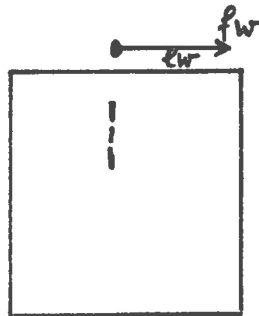
Application is represented as follows:



which corresponds to

$\lambda v. < lv,\ fv, \ldots\ > (w)$

and this may be simplified to



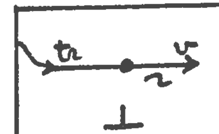
which corresponds to the triple

$< lw,\ fw, ---- >$

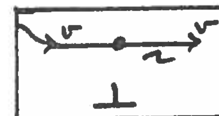
where troughout the dots w is substituted for the free occurrences of v.

Examples

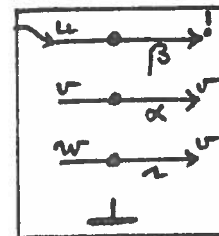
QUOTE $v = \lambda tr < \iota,\ v,\ \perp>$ is represented by



$ID = \lambda v. < \iota,\ v,\ \perp >$ is represented by



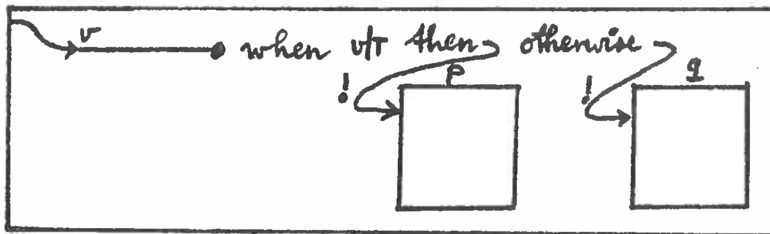
$ASS = \lambda u. < \beta,\ !,\ \lambda v. < \alpha,\ v,\ \lambda w. < \iota,\ v,\ \perp>>>$ by



.2 Operations on processes

The main reason for our representation being not fully formally right is that in general the drawings can only be constructed "at run time", i.e. "dynamical", whereas our notation suggests that it is done "at compile time", i.e. "statically". As long as our notation is read as a "statically formulated recipe for constructing the representation during runtime", the representation presented here is allright. Actually, we will not use the notation in another sense than this one.

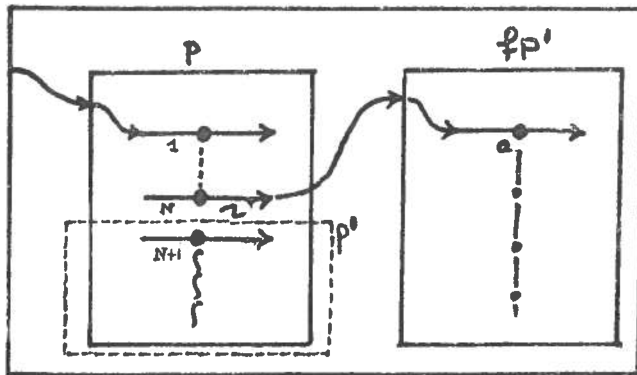
.2a According to the definition COND p q = $\lambda v.\ v/T \supset p!,\ q!$ the conditional is represented by



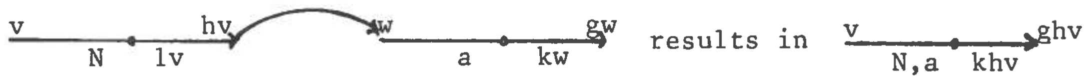
The conditional composition is rather difficult to represent statically. Here one really needs the simplifications due to an application to a particular argument.

.2b EXTEND pf = $\lambda v.\ (pv)_1 = 2 \supset f(pv)_3(pv)_2,\ <(pv)_1,\ (pv)_2,$ EXTEND $(pv)_3 f>$:

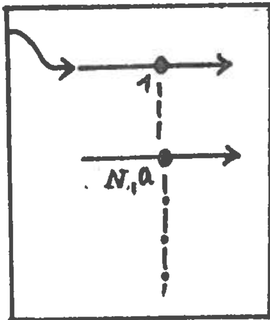
(n.b. $f \in P \rightarrow P$)

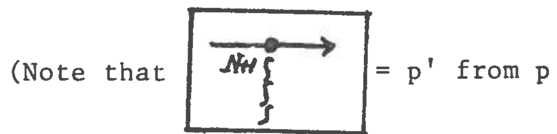


where it is understood that step N of the process p is the dynamical first one which will send a message to the resultlocation ι.
In general, the link from p to fp', and even fp' itself, can only be drawn when all the preceding steps of p are supplied with an argument value.
Due to the link from p to fp' the two steps labelled N and a are melted together into one step, which we will label by N,a:

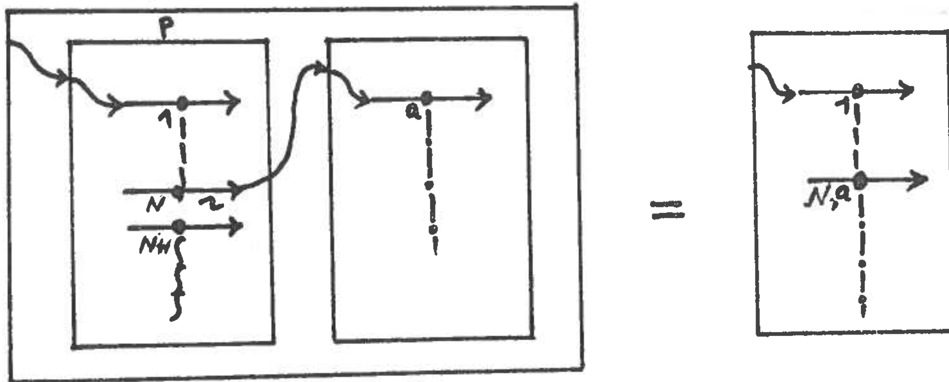


Hence the process resulting from EXTEND pf in the above example equals



(Note that  = p' from p has disappeared. It may however be used in fp').

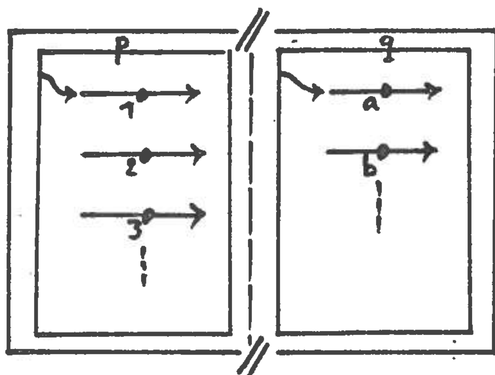
.2c Serial composition p * q = EXTEND p(kq) (where k = λp.q.λp'.q):



Step N being the dynamical first one with location ι.

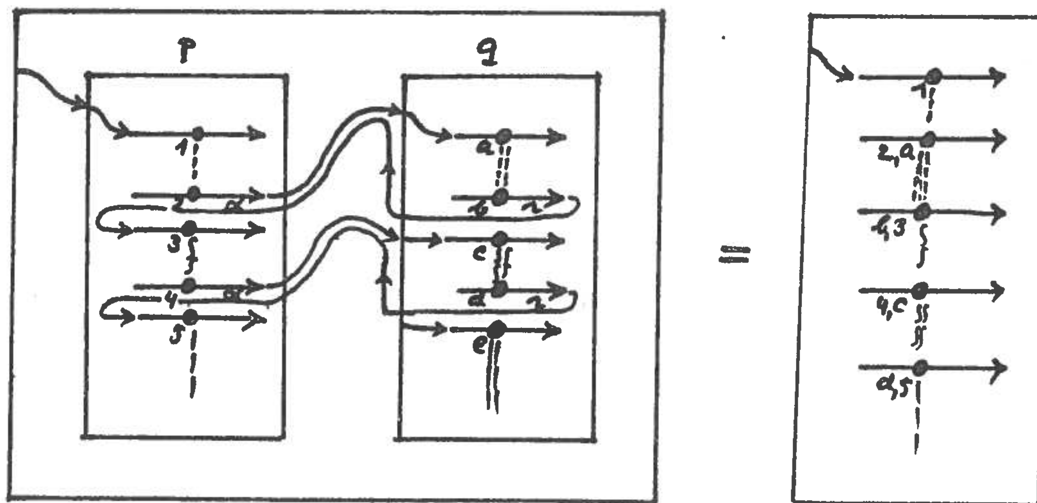
.2d Choise and parallel composition.

Representation by explicit mentioning of the conditional compositions and interrogations of ω will be very awkward. Therefore, the "statically formulated recipe for runtime construction of the picture representation" of the ||-composition is:



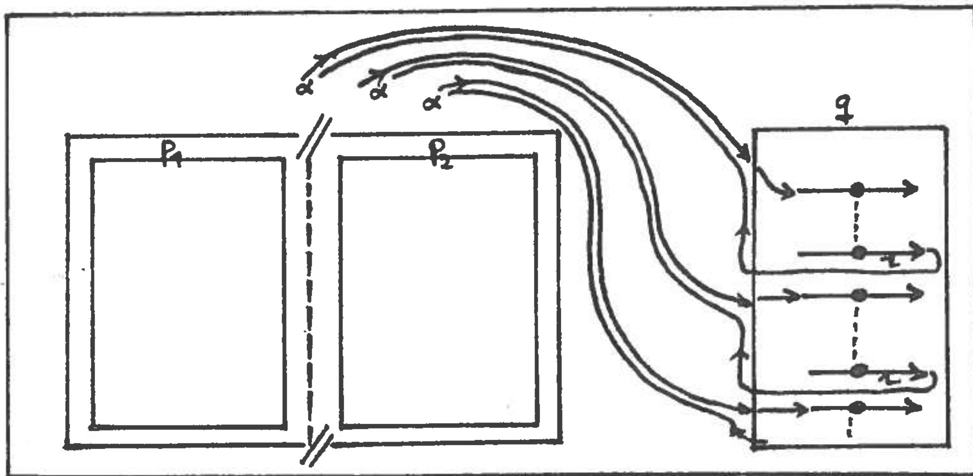
Thus it is understood that the result is an arbitrary interleaving of the steps 1, 2, 3... and a, b,.... and we suppress in this notation all interrogations of the oracle and do not mention the pairing of the intermediate and final results of the separate processes.

.2e The BIND combinator BIND $\alpha pq = \lambda v.\ (pv)_1 = \alpha \supset$ EXTEND $q(\text{BIND}\alpha(pv)_3)(pv)_2$, $<(pv)_1 \cdot (pv)_2$, BIND $\alpha(pv)_3 q\ >$. BIND αpq is represented by



It is the very case of binding where our notation pays off. In particularly we see that the resulting process is indeed a process and that successive parts of q are substituted in p.

Moreover, when the process p makes no more interrogation to α then the rest of q will disappear in the composition BIND αpq.
Of course, the steps 2, 4 and b, d are understood to be the dynamical first, second... ones which will send a value to locations α resp. ι.
When p is a process $p_1||p_2$, then our notation will be



where it is understood that the links to and from q are tied in sequence to the interrogations of α and their successive steps in the $||$-composition.

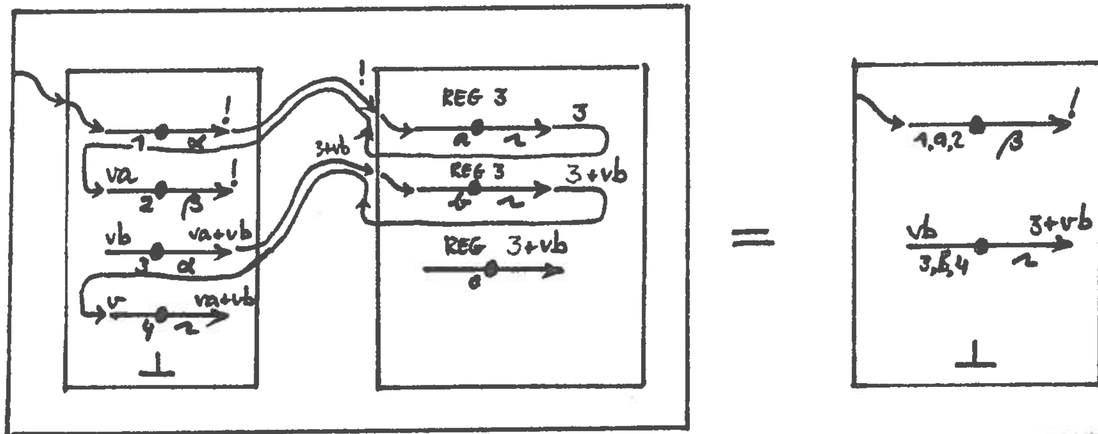
.3 <u>Some examples</u>

Let for the present ⟦ text ⟧ denote the meaning of the text in its environment.

.3a Let $[\![x := x + y]\!] = \lambda tr. < \alpha, !, \lambda va. < \beta, !, \lambda vb. < \alpha, va + vb, \lambda v. < \iota, va + vb, \bot >>>>$.

Let $[\![\underline{reg}\ x = 3\ \underline{in}\ (x := x + y)]\!] = \text{BIND}\, \alpha\, [\![x := x + y]\!]\ \text{REG}\ 3$.

Then, in our notation we get for the latter formula:



i.e. $\lambda tr. < \beta, !, \lambda vb. < \iota, 3 + vb, \bot >>$.

Note that the meaning of the programtext is an interrrogation of the location of y, delivering the value returned incremented by 3. This is the pure extensional meaning, because one can neither see by what device the increment was forced, nor can one detect any remainders of the register which internally was associated with reg x = 3.

.3b Let $[\![:=]\!] = \lambda tr. < rl, !, \lambda rv. < ll, rv, \lambda v. < \iota, rv, \bot >>>$,

let $[\![+]\!] = \lambda tr. < lloc, !, \lambda lval. < rloc, !, \lambda rval. < \iota, lval + rval, \bot >>>$,

let $[\![y]\!] = \lambda v. < \beta, v, \lambda yval. < \iota, yval, \bot >>$,

let $[\![x]\!] = \lambda v. < \alpha, v, \lambda xval. < \iota, xval, \bot >>$.

Then it is reasonable to define the meaning of the text

reg x = 3 in (reg y = 4 in (x : = x + y)) as

BND α BND β BND ll BND rl $[\![:=]\!]$ BND rloc BND lloc $[\![+]\!][\![x]\!][\![y]\!][\![x]\!]$ (REG 4) (REG 3).

Here one really needs machine help in order to evaluate the formula, if one sticks to the λ -calculus formalism ! But in our representation the evaluation is done by drawing merely some links without changing any of the given representations of parts of the formula.

See figure 1.

Again, the meaning is purely extensional and noting is left of the registers which were associated with the locations for x and y.

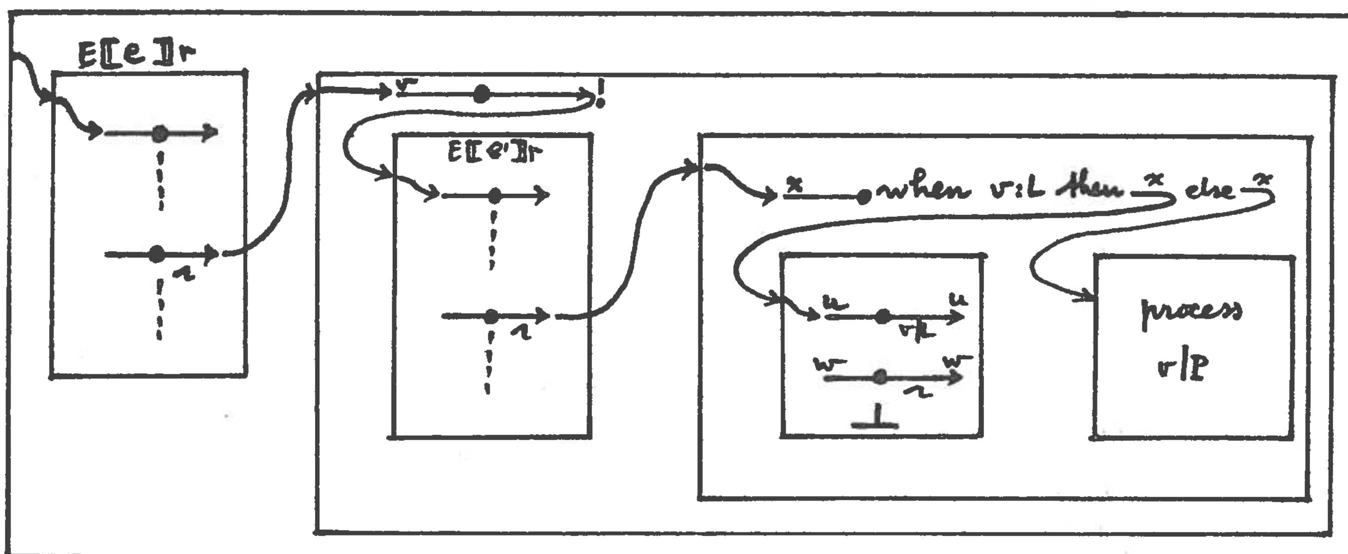
Under the conventions of .3b,

fig 1. The meaning of (reg x=3 in (reg y=4 in (x := x+y))) is:

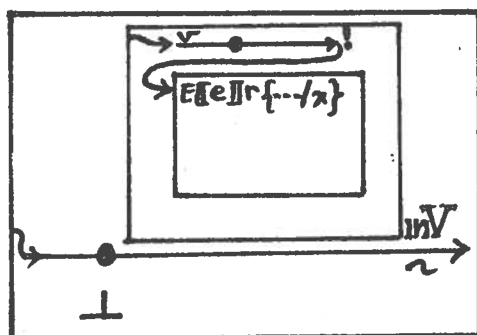
BIND α BIND β BIND ll BIND rl ⟦:=⟧ BIND rloc BIND lloc ⟦+⟧ ⟦x⟧ ⟦y⟧ ⟦x⟧ (REG 4) (REG 3).

.3c The semantics of some language constructs.

$E[\![e(e')]\!]r = E[\![e]\!]r \star \lambda v.\ (E[\![e']\!]r \star (v : L \supset \lambda u < v/L,\ u,\ ID > ,$
$v/P))!:$



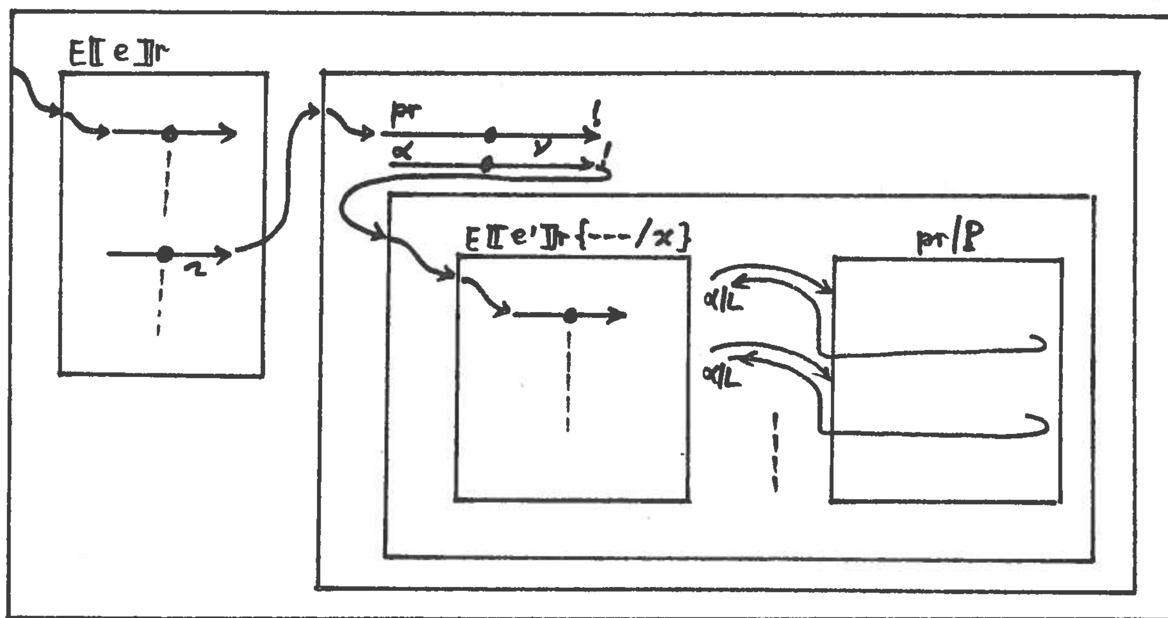
$E[\![\pi x.e]\!]r = QUOTE\ (\lambda v.E[\![e]\!]r\{(QUOTE\ v)/x\}!)inV:$



(the dots abbreviate (QUOTE v))

$E[\![$ <u>let</u> <u>slave</u> x <u>be</u> e <u>in</u> e'$]\!]r =$

$E[\![e]\!]r \star \lambda pr.<\ \mathbf{v},\ !,\ \lambda\alpha.\ BIND\ \alpha/L\ \ E[\![e']\!]r\{(QUOTE\ \alpha)/x\}\ \ pr/P\ !\ >:$



Note that pr/P disappears when in E[e']r{.../x} no interrogation will be made to α/L.

PART 2 Defect of the semantics

There is a serious designmistake in the definition of the language. We emphasize that we do not claim the semantics being inconsistent. What we will point out is that the semantic definition is not the intended one and in particular the program for adding up the numbers from 1 to 100 in parallel does not work as in intended.

An attempt is made to define the sematics as intended but the concepts involved tend to be more "operational" objects than "mathematical" such as processes and functions. Therefore the attempt doesnot satisfy me.

.1 The effect of binding, the defect of the semantics.

.1a Let p be a process in which no step will make an interrogation to β.
Then BIND βpq = p.
This causes that the programtext

<u>let</u> <u>slave</u> settoone <u>be</u> πx.x : = 1 <u>in</u>
<u>let</u> <u>slave</u> y <u>be</u> Cell (0) <u>in</u> settoone (y)

has not as meaning the trigger process
$\lambda tr. < \iota, 1, \perp>$
but is instead the process
$\lambda tr. < \beta/L, 1, \lambda v. < \iota, 1, \perp >>$
where β is the second value delivered by the generator process bound to ν.
Indeed, the slave y is local to the scope of slave settoone and is therefore bound first. In that binding no interrogation of its corresponding location (assumed to be β/L) is made. At the time when ⟦πx.x : = 1⟧ is bound to the location (say α/L) for settoone this process recieves β as value and then sends the value 1 to location β/L .
But this interrogation is left open: the vacuus binding of ⟦Cell(0)⟧ to β/L has been done sometime ago.
For the picture representation see fig. 2, page 13. Formally we get

(1) E⟦programtext⟧r =

(2) E⟦πx.x : = 1⟧r $\star$ λsto. < ν , !, $\lambda\alpha$.

(3) BIND α/L E ⟦..⟧r{(QUOTE α)/settoone} sto/P ! >,

(4) E⟦...⟧r{(QUOTE)/settoone} =

(5) E⟦Cell (0)⟧r $\star$ λreg. < ν, !, $\lambda\beta$.

(6) BIND β/L E⟦settoone (y)⟧r{(QUOTE α)/settoone , (QUOTE β)/y}
reg/P ! >,

(7) E⟦settoone (y)⟧r{(QUOTE α)/settoone , (QUOTE β)/y} =

(8) = --- = λtr. < α/L, β, ID >.

Now, in (8) and so in (7) no interrogation of β/L is made. Therefore (6) reduces to (7) and so to (8). Hence (5) and (6) together reduce to λtr. < ν, !, $\lambda\beta$. < α/L, β, ID >>, and so does (4). Moreover, it is easy to check that in (3) the sto/P is the process λloc. < loc/L, 1, ID > , hence (3) reduces to λtr. < ν, !, $\lambda\beta$. < β/L, 1, ID>>. Thus (2) and (3) together reduce to λtr. < ν, !, $\lambda\alpha$. < ν. !, $\lambda\beta$. < β/L, 1, ID>>>, and so does (1). After binding a generator to ν this results in λtr. < β/L, 1, ID>.

.1b This means that in general the sematics does not give the intended process when some outer-declared slave is <u>applied to</u> some inner declared one.

This is rather unsatisfactory. In fact, the program in the paper of Milner fails by this reason. The outer-declared slave is INC, the inner one is j. Thus we can make the following changements in [1], page 14 in the last but one sentence:

with these⟹ with more; cannot⟹ can; is correct !⟹ is not correct !!.

Just declaring the appropriate slaves local to the declaration of their actual parameters seems to solve the problem, but conflicts the intention of being one unique indivisible process: in general one outer slave has to be declared several times as an inside one, in particular this may be the case in a parallel composition.

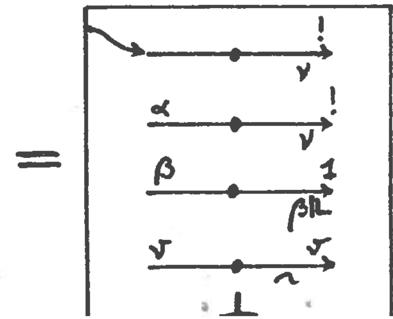
(2) (3) (5) (6) (7) (6) ( ) (the numbers refer to 1a)

$E[\![\pi x. x:=1]\!] r * \lambda sto.\langle \nu, !, \lambda\alpha.\ BND\ \alpha|L \left(E[\![Cell(0)]\!] r \{\frac{Q.\alpha}{settoone}\} * \lambda reg.\langle \nu, !, \lambda\beta.\ BND\ \beta|L\ E[\![settoone(y)]\!] r \{\frac{Q.\alpha}{settoone}, \frac{Q.\beta}{y}\} \right) reg|P\ !\rangle\ sto|P\ !\rangle$

According to the semantics of Milner's paper,

fig 2 The meaning of <u>let slove</u> settoone <u>be</u> $\pi x. x:=1$ <u>in</u> <u>let</u> <u>slove</u> y <u>be</u> Cell(0) <u>in</u> settoone(y)

is $\lambda tr.\langle \nu, !, \lambda\alpha.\langle \nu, !, \lambda\beta.\langle \beta|L, 1, \lambda v.\langle r, v, \perp\rangle\rangle\rangle\rangle$ =



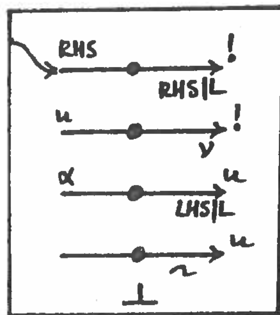
Nevertheless I do see some sense in the proposed semantics. The program writer of an outer block has an inherent protection of his slave processes against being un(?)-intentionally used by writers of inner blocks. This is quite similar to some proposals of Dijkstra, I recently heard of. But no facility, like a parameterspecification, seems to be possible to allow an inner block program writer use the outer declared slave processes. Indeed, such a facility would solve the "designmistake".

.2 More comments on the program

The convention of letting x : = y be an abbreviation of z(val(y)) is not consistent with the convention of page 5, viz.letting x : = y be an expression whose value is the value assigned, i.e. y. Indeed, by the semantic definition the value of an application x(...y...) is the value returned by the process corresponding with x. A better alternative is letting x : = e be an abbreviation of : = (x)(e), where : = is declared by

```
let rec : = be  ( πlhs. πrhs. let slave r be Cell (val(rhs)) in
                  lhs (val(r)); val (r))
```

In figure 3 the representation of r(: =) according to part 1 of this paper is given. In that figure the innermost box postfixed with inV is easily seen to be equal to



, i.e. $\lambda RHS. < RHS/L, !, \lambda u. < \nu, !, \lambda\alpha. < LHS/L, u, \lambda tr. < \iota, u, \perp>>>>$.

This convinces me that r(: =) now has the intended meaning.

My personal experiences in deriving the above definition for : = where as follows. I made the following attempts:

```
let rec : = be πlhs.πrhs.lhs (val (rhs)); val (rhs)
let rec : = be πlhs.πrhs. let rec r be val (rhs) in lhs (r); r
let rec : = be πlhs.πrhs, let slave r be Cell (val (rhs)) in lhs (r); r
let rec : = be πlhs.πrhs. let slave r be Cell (val (rhs)) in lhs (r); r( )
```

E⟦π. lhs. π rhs. let slave r be Cell(val(rhs)) in lhs(val(r)); val(r) ⟧r

RHS !

E⟦let slave r be Cell(val(rhs)) in lhs(val(r)); val(r) ⟧r{QUOTE LHS/lhs, QUOTE RHS/rhs}

E⟦Cell(val(rhs))⟧r{Q.LHS/lhs, Q.RHS/rhs}

! RHS/L

REG u

u inV r

⊥

This process is already simplified as much as possible under the assumption "isL(RHS)".

rpr ! v

dr !

BND dr|L E⟦lhs(val(r)); val(r)⟧r{QUOTE LHS/lhs, QUOTE RHS/rhs, QUOTE dr/r} λpr|P

E⟦lhs(val(r)); val(r)⟧r{··/.., ··/.., ··/..}

! dr/L

v v LHS/L

! dr/L

w w r

⊥

This process is already simplified as much as possible.

dr dr

rpr|P

REG u r

REG u r

REG u

inV r

⊥

inV r

⊥

fig. 3. The meaning of :=, i.e. r(:=), within the scope of let rec := be (π lhs. π rhs. let slave r be Cell(val(rhs)) in lhs(val(r)); val(r))

until I realised that each indentifier has as meaning a quoting process which

- either yields the <u>location</u> of the intended process (<u>slave</u> identifiers),
- or <u>immediately</u> yields <u>the intended process</u> (<u>rec</u> identifiers).

As a consequence

- <u>slave</u> identifiers are used senseless when they stand alone (as the last but one symbol of the program:
  it should be val(s) in stead of s). By parameter transmission they ultimately have to be used as "function symbols" of an application, because it is the semantic equation of that very language construct which uses the delivered location in the right way.
- <u>rec</u> identifiers need not be function symbols. In particular we can make the program looking more natural by deleting both the abstraction πZ. and all arguments ( ) of Add. In addition, in stead of repeating two times val (r) in the declaration of : = mentioned above, we can replace it by one identifier v provided we declare <u>let</u> <u>rec</u> v <u>be</u> val (r).

.3 <u>An attempt to repair the semantics</u>

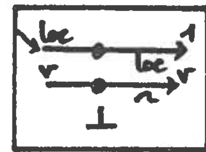
.3a Intuitively.

We should like to delay the binding of slave processes until the whole programtext is considered, just as only at the end the generator is bound to ν by means of MNG.

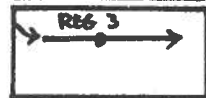
More precisely, all nested slave declarations should be evaluated yielding processes which are pushed into a list of processes.

The head - or tail - of the list is the process corresponding to the main body of the program. To the list, being the evaluation of the complete programtext, a binding combinator is applied, yielding again a process in which there only should occur interrogations of ν and ω.

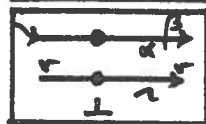
In my picture representation as follows:

let slave settoone be x.x : = 1 in 

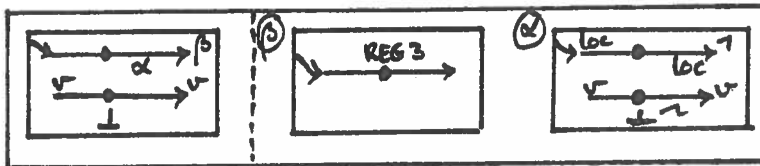
and

let slave y be Cell (3) in 

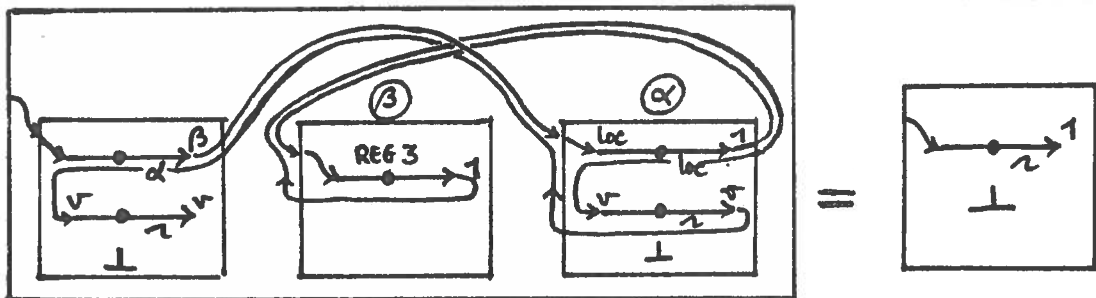
and

settoone (y)  .

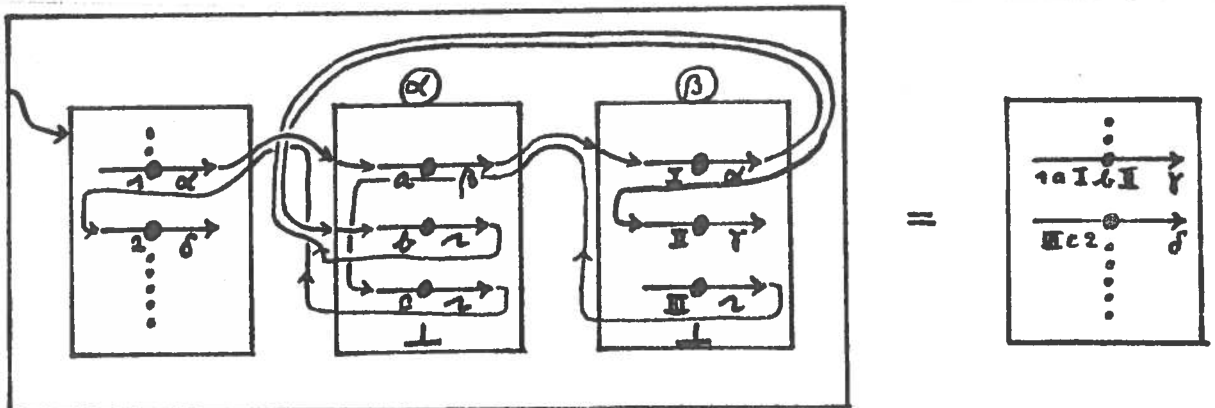
And the composition of these three should be the list



which after application of the binding combinator should result in



Moreover, the bindings should force the processes to interact like a kind of coroutines: after each interrogation (link into the process), the process answers (link from the process) upon the next result flag the most recent process which has not yet recieved an answer. A typical example is



Note that in this way slave processes may be declared recursively: an interrogation of itself is allowed.

.3b Formally.

Whether I do or do not succeed in giving an elegant formal treatment in the λ - notation, the above exposition should be convincing of the consistency of the ideas. It remains to be seen,however, whether such a formalization indeed reflect the intended semantics. I strongly believe that the above approach solves the problem to which this part of the paper was devoted. But in stead of the very elegant semantic objects, viz. merely processes, we are now led to complex objects being lists of pairs of locations with processes. I find this very unsatisfactory, but at the moment I see no way to overcome it. In fact the obvious way of defining the binding combinator requires the lists being lists of triples: locations, stacks of locations (the return addresses) and processes. Due to my unsatisfaction I do not present the formal definition of the binding combinator and the required domain equations. Instead, I announce that future work will be looking for a more elegant solution to the problem, preferably sticking to merely processes.

I acknowledge critical comments by Wiek Vervoort on a draft of this paper.

References

R. Milner:"An Approach to the semantics of parallel programs" Edinburgh Techn. Memo, Univ. of Edinburgh (1973).