stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 192/82 MAART

A.P.W. BÖHM & A. DE BRUIN

DYNAMIC NETWORKS OF PARALLEL PROCESSES

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68D99, 68F20, 68B10

1982 CR Categories: C.1.3, F.1.2, F.3.2, D.2.4

Dynamic networks of parallel processes *)

by

Wim Böhm **)
Arie de Bruin

ABSTRACT

In this paper we investigate a model of parallel computation based on Kahn's simple language for parallel programming [KAHN74]. We present two sorting algorithms and a matrix multiplication algorithm and prove them correct.

KEY WORDS & PHRASES: parallellism, data flow, Kahn's language, parallel coroutines, denotational semantics, program proving, sorting, matrix multiplication

---

*) This report will be submitted for publication elsewhere.

**) Vakgroep informatica, Rijksuniversiteit Utrecht, Princetonplein 5, Utrecht, The Netherlands

1. THE LANGUAGE

1.1. Informal description

A program consist of a number of process declarations and a main body. In the main body processes are initiated and connected together into a process network by means of channels. The processes communicate with each other via these channels only. Channels are queues of messages (also called tokens or values). The processes look very much like coroutines.

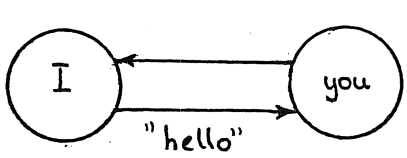
Each process declaration consists of a heading and a body. In the heading formal channels are declared, specifying the name, whether the channel is an input channel or an output channel, and the type of the tokens travelling on the channel. Apart from formal channels, formal value parameters can occur in the heading. The body of a process declaration consists of ordinary data declarations, control structures, assignment statements and two communication statements: read and write. A process can read (consume a value) from an input channel and write (produce a value) on an output channel. If a channel is empty when the consuming process performs a read on it, the consuming process is blocked until the producing process has written a value on the channel.

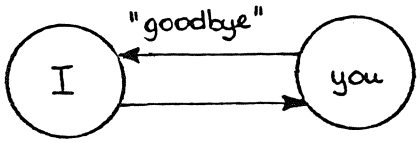
So the model looks very much like the coroutines of e.g. SL5 [SL5D1a] except that when a process writes a value on one of its output channels ("resumes an environment" in SL5 terminology) it is not suspended but carries on in parallel with the process it has activated. The model can also be viewed as a data driven model or as a history level data flow model.

EXAMPLE

```
(process I(string in ear string out mouth):
   (string l;
    repeat read(ear,l); write(mouth,"goodbye")
    forever
   );

process you(string in ear, string out mouth):
   (string l;
    repeat write(mouth,"hello"); read(ear,l)
    forever
   );

main I(line1,line2) || you(line2,line1)
)
```

The above program shows one instance of an "I" process declaration and one instance of a "you" process declaration, communicating via two string channels line1 and line2. In this particular example there will never be more than one string on the channels:





1.2. Semantics

The semantics of parallel processes like the above has been sketched in [KAHN74]. Each process specifies a function which takes input histories as arguments and yields output histories. A history is a (possibly infinite, possibly empty) sequence of values. The input and

output histories are meant to model the sequences of values that travel on the channels. For instance the process "I" described above determines a function f taking an input history consisting of strings and yielding an output history of the same type. More formally, we have f:D->D where D is the domain of all sequences of strings. We define f(X) to be the sequence consisting of strings "goodbye" with the same length as X. Notice that f(X) does not depend on the contents of X but only on its length. Similarly we have the function g:D->D where g(Y) is the sequence of strings "hello" which has the same length as Y.

In general it will not be possible to give a straightforward definition of such functions. Often these definitions will be recursive. The recursive version of the definition of f is:

$$f(X) = \begin{cases} \langle\rangle, \text{ if } X = \langle\rangle \\ \langle\text{"goodbye"}\rangle \hat{} f(R(X)), \text{ if } X \neq \langle\rangle \end{cases}$$

where $\langle\rangle$ denotes the empty sequence, $\hat{}$ denotes concatenation of sequences, R(X) denotes the sequence of all elements of X except the first one. We also have the function F(X) which takes the first element from X. So in general we have $X = F(X)\hat{}R(X)$ for non-empty X.

The output that a process yields is generally not only dependent on its input but also on the value of its local variables. The function describing the meaning of the process will therefore often have an extra parameter giving the relevant part of the internal state of the process .

Suppose, by way of example, that the process "you" above does not only say "hello" but also tells how many times it has said "hello". The function corresponding to this process can then be defined as follows:

$$f(count,X) = \begin{cases} \langle\rangle, \text{ if } X = \langle\rangle \\ \langle\text{"hello"},string(count+1)\rangle \hat{} f(count+1,R(X)), \\ \qquad \text{otherwise.} \end{cases}$$

(string(val) converts an integer value to an english word).

The meaning of the new process "you" is given by the function g defined by

$$g(X) = f(0,X)$$

The functions corresponding to processes have several properties and we will discuss some of these now.

Histories can be ordered according to the information they contain. We say that history Y contains more information than history X, notation $X \sqsubseteq Y$, if X is a prefix (not necessarily a proper prefix) of Y. Now it can immediately be seen that functions which are meanings of processes are monotonic, that is (restricting ourselves to the case of one input and one output channel): if $X \sqsubseteq Y$ then $f(X) \sqsubseteq f(Y)$. For a process takes its input values one by one from the input channel and its actions are completely determined by these values. So if input history X is a prefix of input history Y, then the process will act identically on the common prefix and thus generate the same values on the output channel. The remaining input on input history Y can only have the effect that more values will be added to the output history.

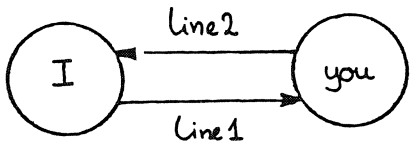
The next property, continuity, has something to do with approximating an infinite sequence by its finite prefixes. These prefixes form a chain, that is a row of histories $X_1, X_2, X_3, \ldots$ such that $X_i \sqsubseteq X_{i+1}$ for every i. Now every such chain has a least upper bound $\sqcup X_i$. This is so because either the chain is stable, that is the chain has a tail $X_k \sqsubseteq X_{k+1} = X_{k+2} = X_{k+3} = \ldots$ and then $X_i = X_{k+1}$, or the chain is not stable but then every element X_i has length greater than a predecessor X_{i-n} and $\sqcup X_i$ will be the infinite history X with the property that all X_i from the chain are a prefix of X.

Now suppose that a process P yields, when given an infinite input history X, an infinite output history f(X). Consider the values $f(X_1)$, $f(X_2), \ldots$ where f is the function associated with P and $X_1 \sqsubseteq X_2 \sqsubseteq X_3 \ldots$ is the row of all finite prefixes of X (thus $X = \sqcup X_i$). These values form a chain (by monotonicity of f), that is we have $f(X_1) \sqsubseteq f(X_2) \sqsubseteq$

$f(X_3) \sqsubseteq$ ... Now the crucial observation is that for every element x of f(X) we have that at the moment it is generated only a finite number of elements of X have been read. Furthermore, the whole sequence up to x must have have been generated. This means that an arbitrary finite approximation of the output history f(X) is given by letting process P work on a finite input history $X_k \sqsubseteq X$. In other words: for all m there is a k such that $f(X_k)$ has length $\geq$ m and $f(X_k) \sqsubseteq f(X)$. And this is equivalent to saying that $f(X) = f(\sqcup X_i) = \sqcup f(X_i)$. Now the latter equality is the definition of continuity.

So we can conclude that every process P has a function f associated to it which is monotonic and continuous.

Finally we will discuss the meaning of a network. IN [KAHN74] this meaning is defined as the histories on all channels. The meaning of the network



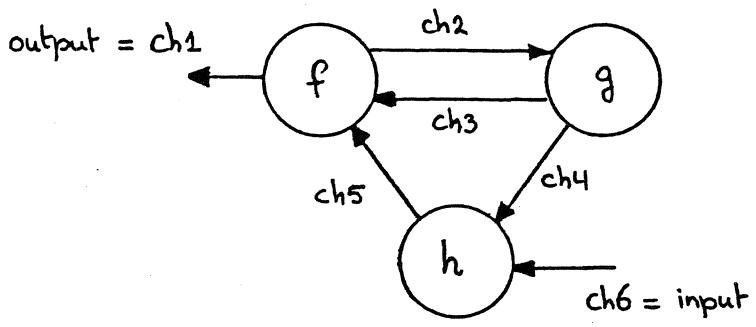
given before will thus be a pair of histories <LINE1,LINE2> where LINE1 and LINE2 contain an infinite number of "hello"s and "goodbye"s respectively.

In general one can obtain a set of equations from a network specification and the functions corresponding to the processes in the network. These equations specify what the values of the histories are to be. In the above example we have

$$Y = f(X)$$
$$X = g(Y)$$

where X,Y are the histories on line1 and line2 respectively. These equations have LINE1 and LINE2 as solutions.

We now present a more intricate case:



We associate with this network the set of equations

$$\langle X_1, X_2 \rangle = f(X_3, X_5)$$
$$\langle X_3, X_4 \rangle = g(X_2)$$
$$X_5 = h(X_4, X_6)$$

The fact that functions f, g, and h are continuous guarantees that there is a solution in $X_1, \ldots, X_6$ of these equations. Moreover it can be proved that executing the programs in the processes results in histories that satisfy these equations. So the meaning of the program is rightly described by these equations. In subsequent sections we will prove properties of programs using only the fact that the histories on the channels satisfy equations such as the ones above.

1.3. The dynamic model

Up till now a process declaration contains ordinary statements dealing with the internal memory of the process and communication statements to manipulate its input and output channels. To make the model essentially more powerful than a sequential model we add an expand statement.

The expand statement replaces the process in which it occurs by a (sub)network of processes. This subnetwork is connected to the rest of the network via exactly the same input and output channels as the old process was. Expansion can be compared with SL5 filters [SL5D7a]. SL5 allows only linearly shaped networks, while we allow all kinds of networks.

When an expand statement is executed the following things happen:

- the old process is disconnected from the network; its channels are temporarily closed on the side of the expanding process.
- new processes are created
- the newly created processes and possibly the old process (which initiated the expansion) are connected by internal channels such that they form a subnetwork
- the subnetwork is connected to the rest of the network via the temporarily closed channels
- the new processes start computing in their initial state; the old process will proceed after the expand statement (if it is part of the subnetwork).

The rest of the network will carry on computing while an expansion takes place.

An expand statement consists of a number of create parts and zero or one keep part. In the create parts new processes are called and their actual channel parameters and actual value parameters are specified. In the keep part the old process is called. The syntax:

expand statement:
   expand,
      create parts ,(comma, keep part),
      (comma, create parts),
   expand.

create parts:
   create part;
   create part, comma, create parts.

create part:
   create process name with channel identifications,
      (comma, value identifications),

keep part:

keep process name with channel identifications.

channel identifications:

channel identification;

channel identification, comma, channel identifications.

channel identification:

formal channel name = actual channel name.

value identifications:

value identification;

value identification, comma, value identifications.

value identification:

formal value name = expression.

The newly created actual channels will occur twice, once as an input channel and once as an output channel. The old channels will occur only once and their type will not change. The process name in a keep part is the name of the process in which the expansion occurs.

So we have the following scheme;

```
process P(in in1,...,inn out out1,...,outm):
   (  :
      .
    expand
      create.....
      create.....
      keep.......
    expand.
      :
      .
      .
   )
```

Now suppose f is the function corresponding to the process started just before the expand statement is executed. Thus f takes an n-tuple of histories as argument and yields an m-tuple as result. Now the effect of the expanding process on its histories must be the same as the effect of the network into which it expands. That is, we have

$$f(X_1,\ldots,X_n) = (Y_1,\ldots Y_m)$$

where

$$\begin{aligned} Y_1 &= \ldots \\ Y_2 &= \ldots \\ &\vdots \\ Y_m &= \ldots. \end{aligned} \qquad (*)$$

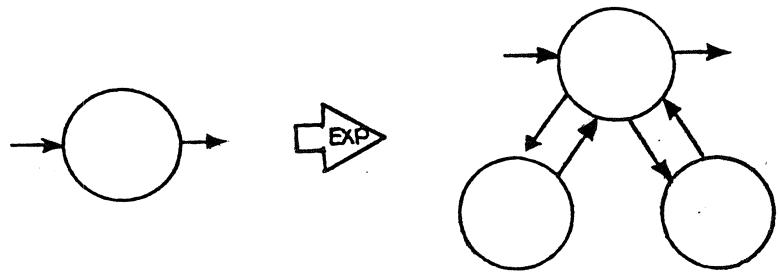
The output histories $Y_1,\ldots,Y_m$ are a solution of the equations (*) derived from the network. Moreover it can be proved that it must be the smallest solution of (*).

The right hand sides of these equations have the form $g(Z_1,\ldots,Z_k)$. So we have to determine these functions g. The processes occurring in the network are specified by the expand statement. We have the functions corresponding to the create parts to our disposal. The function corresponding to the keep part can be derived from the meaning of process P when started just after the expand statement with the values of the local variables as they were at the moment of expansion.

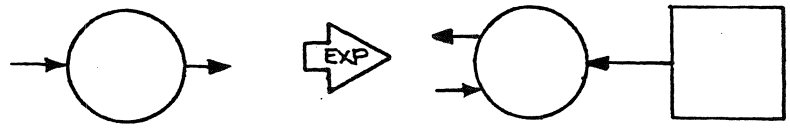
The meaning of the whole process P can be given as a composition of two parts. The second part is given by the function f. The first can be derived from the program text between the beginning and the expand statement.

2. THE ALGORITHMS

We now present the algorithms, which are of course what it's all about. The algorithms were conceived graphically. One thinks in pictures such as



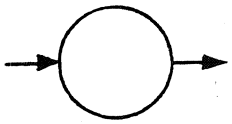
or



The pictures translate easily into process declarations. They also help finding the functions.

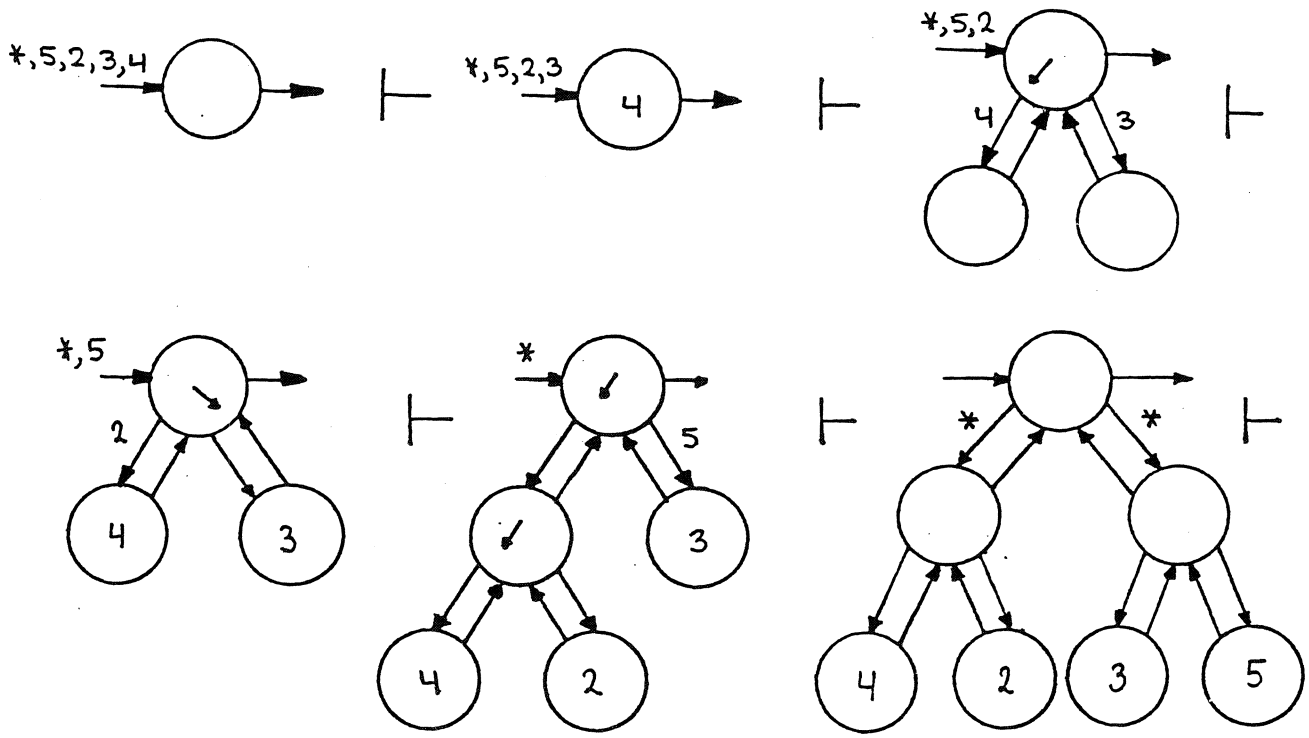
2.1. Sorting in a tree.

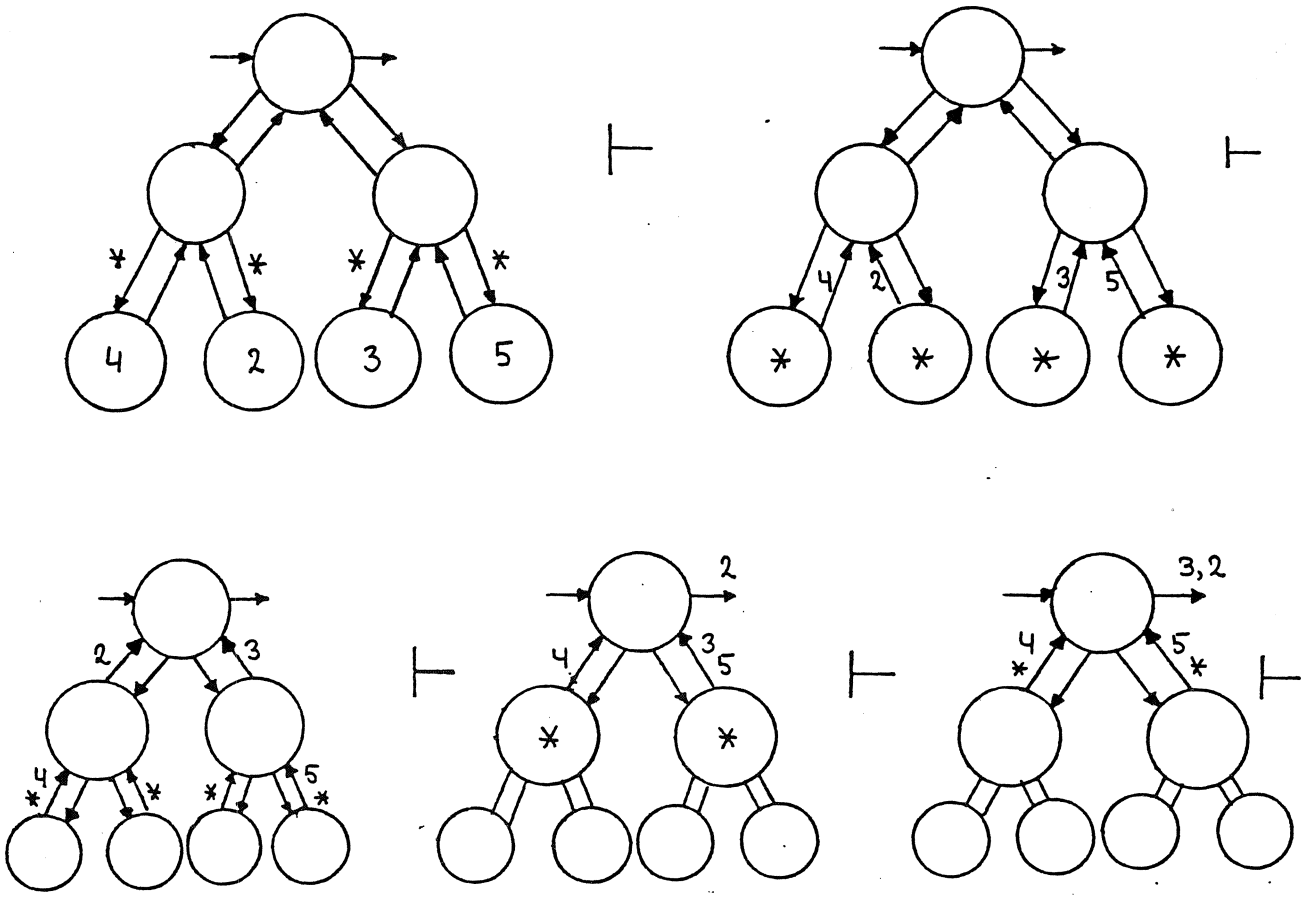
This algorithm is a straightforward adaptation of mergesort. The input consists of a finite sequence of integers followed by an end of file token (*). The input is read into the root of a perfectly balanced `tree`:



When the root gets more than two items it expands into a binary tree. The root now divides the input over its two subtrees. The subtrees act in the same way the whole tree does. When the * is encountered, the root sends it down to both its subtrees. The * travels down the tree until the leaves of the tree are reached. Then the whole tree starts merging the elements upwards.

The pictures below show what happens when the sequence 4,3,2,5,* enters the network.





2.1.1. The program

```
(process sort(int in u int out s):
   (/* terminal node , u unsorted, s sorted */
    int x; read(u,x);
    if x=* then write(s,*)
    else int y; read(u,y);
         if y=* then write(s,x); write(s,*)
         else expand create sorn with u=u, ls=lup, rs=rup, s=s,
                                      lu=ldwn, ru=rdwn, i1=x, i2=y
                   create sort with u=ldwn, s=lup
                   create sort with u=rdwn, s=rup
              expand.
```

```
        if.
    if.
   );

process sorn(int in u,ls,rs int out s,lu,ru int i1,i2):
   (/* nonterminal node */
    int x; bool left=true, right=false; bool dir:=left;
    write(lu,i1); write(ru,i2);
    /* spread the unsorted input u over lu and ru */
    while read(u,x); x≠* do
       if dir=left then write(lu,x); dir:=right
                   else write(ru,x); dir:=left
       if.
    do.;
    write(lu,*); write(ru,*);
    /* merge the sorted inputs ls and rs */
    int l,r;
    read(ls,l); read(rs,r);
    repeat if l=* then /*flush rs */
                  while r≠* do write(s,r); read(rs,r) do.
       elif r=* then /* flush ls */
                  while l≠* do write(s,l); read(ls,rl) do.
       elif l<r
            then write(s,l); read(ls,l)
            else write(s,r); read(rs,r)
            if.
       if.
    until l=r=* ;
    write(s,*)
   );

main
   int in unsorted;
   int out sorted;
```

sort(unsorted, sorted)
)

2.1.2. The functions

We associate with process sort the function f_0 and with process sorn the function f_1. f_1 takes six arguments: three input histories, the values of two input parameters and an extra argument, namely the current value of dir. We have

$f_0(\langle * \rangle \hat{} X) = \langle * \rangle$
$f_0(\langle a,* \rangle \hat{} X) = \langle a,* \rangle$
$f_0(\langle a,b \rangle \hat{} X) = f_1(X,Y,Z,a,b,\underline{true})\downarrow 3,$
where $Y = f_0(f_1(X,Y,Z,a,b,\underline{true})\downarrow 1)$
$Z = f_0(f_1(X,Y,Z,a,b,\underline{true})\downarrow 2)$

$f_1(U,LS,RS,x,y,dir) = (\langle x \rangle,\langle y \rangle,\langle \rangle) \hat{} f_2(U,LS,RS,dir)$

$f_2(\langle * \rangle \hat{} U,LS,RS,dir) = (\langle * \rangle,\langle * \rangle,\langle \rangle) \hat{} f_3(U,LS,RS)$
$f_2(\langle a \rangle \hat{} U,LS,RS,dir) =$
if dir = true then $(\langle a \rangle,\langle \rangle,\langle \rangle) \hat{} f_2(U,LS,RS,\neg dir)$
else $(\langle \rangle,\langle a \rangle,\langle \rangle) \hat{} f_2(U,LS,RS,\neg dir)$

$f_3(U,\langle * \rangle \hat{} LS,\langle * \rangle \hat{} RS) = (\langle \rangle,\langle \rangle,\langle * \rangle)$
$f_3(U,\langle * \rangle \hat{} LS,\langle a \rangle \hat{} RS) = (\langle \rangle,\langle \rangle,\langle a \rangle) \hat{} f_3(U,\langle * \rangle \hat{} LS,RS)$
$f_3(U,\langle a \rangle \hat{} LS,\langle * \rangle \hat{} RS) = (\langle \rangle,\langle \rangle,\langle a \rangle) \hat{} f_3(U,LS,\langle * \rangle \hat{} RS)$
$f_3(U,\langle a \rangle \hat{} LS,\langle b \rangle \hat{} RS) =$
if a<b then $(\langle \rangle,\langle \rangle,\langle a \rangle) \hat{} f_3(U,LS,\langle b \rangle \hat{} RS)$
else $(\langle \rangle,\langle \rangle,\langle b \rangle) \hat{} f_3(U,\langle a \rangle \hat{} LS,RS)$

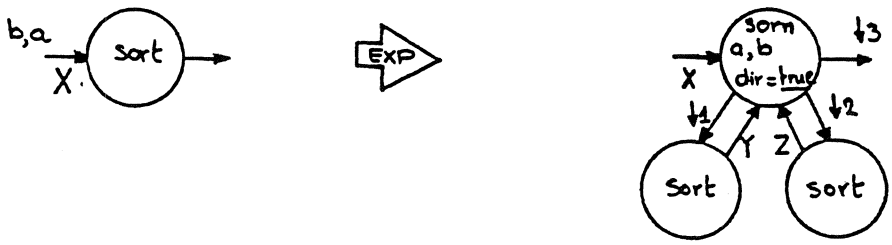
REMARKS

- In the above equations all a and b are unequal *.
- We define the $\downarrow$-operator by $(X_1,\ldots,X_n)\downarrow i = X_i$.
- We define the simultaneous concatenation operator ^ by

  $$(X_1,\ldots,X_n) \hat{} (Y_1,\ldots,Y_n) = (X_1\hat{}Y_1,\ldots,X_n\hat{}Y_n)$$

- Any function applied to arguments of a form not specifed in the above equations yields a tuple of empty histories. For instance

  $f_2(<>,LS,RS,dir) = (<>,<>,<>)$ for all LS, RS and dir.
- The third eqation describes the effect of the expand statement in sort. It corresponds to the following picture



- In the equation defining f_1, one observes that f_2 does not depend on x and y any more. This is a direct consequence of the fact that the process sorn outputs these values immediately and does not use them any more.
- The functions written here are streamlined versions of the functions one would derive by a literal translation fom the program text. We will comment on this later on.

2.1.3. The proof.

Theorem Let $X = D\hat{}<*>\hat{}X'$ with D finite and not containing *.
Then $f_0(X) = D'\hat{}<*>$ where D' is an ordered permutation of D.

This theorem will be proved using the following lemma's:

Lemma 1 [behaviour of f_3]. Let $Y = D'\hat{}\langle * \rangle\hat{}Y'$ and $Z = D''\hat{}\langle * \rangle\hat{}Z'$ with D' and D'' finite and not containing *. Then $f_3(X,Y,Z) =$
$= (\langle\rangle,\langle\rangle,D\hat{}\langle * \rangle)$ where D is a permutation of $D'\hat{}D''$.
Moreover if D' and D'' are ordered, then so is D.

Proof. Immediate by induction on $|D'|+|D''|$

Lemma 2 [behaviour of f_2]. Let $X = D\hat{}\langle * \rangle\hat{}X'$ with D finite and not containing *. Then $f_2(X,Y,Z,dir)\downarrow i = D_i\hat{}\langle * \rangle$ for $i = 1,2$,
where $D_1\hat{}D_2$ is a permutation of D, and $f_2(X,Y,Z,dir)\downarrow 3 =$
$= f_3(X',Y,Z)\ 3$.

Proof. By induction on $|D|$.

Basis $|D|=0$. Then we have $X = \langle * \rangle\hat{}X'$ and by definition of f_2:
$f_2(X,Y,Z,dir) = (\langle * \rangle,\langle * \rangle,\langle\rangle)\hat{}f_3(X',Y,Z) =$
which equals by Lemma 1: $(\langle * \rangle,\langle * \rangle,\langle\rangle)\hat{}(\langle\rangle,\langle\rangle,f_3(X',Y,Z)\downarrow 3)$.

Induction step . $|D|>0$ implies $X = \langle a\rangle\hat{}D''\hat{}\langle * \rangle\hat{}X'$.

Suppose without loss of generality that dir = true.

Then $f_3(X,Y,Z,dir) = (\langle a\rangle,\langle\rangle,\langle\rangle)\hat{}f_2(D''\hat{}\langle * \rangle\hat{}X',Y,Z,\ dir) =$

(by induction)

$$= (\langle a\rangle,\langle\rangle,\langle\rangle)\hat{}(D_1\hat{}\langle * \rangle,D_2\hat{}\langle * \rangle,f_3(X',Y,Z)\downarrow 3) =$$
$$= (\langle a\rangle\hat{}D_1\hat{}\langle * \rangle,D_2\hat{}\langle * \rangle,f_3(X'Y,Z)\downarrow 3),$$

with $D_1\hat{}D_2$ a permutation of D''. But this implies that $\langle a\rangle\hat{}D_1\hat{}D_2$ is a permutation of D.

Proof of the theorem. By induction on the length of D.
The basic cases $|D| = 0$ and $|D| = 1$ are straightforward.
So suppose $|D| \geq 2$, that is $D = \langle a,b\rangle\hat{}D''\hat{}\langle * \rangle\hat{}X'$.
Then $f_0(X) = f_1(D''\hat{}\langle * \rangle\hat{}X',Y,Z,a,b,\underline{true})\downarrow 3$

where $Y = f_0(f_1(D''\hat{}\langle * \rangle\hat{}X',Y,Z,a,b,\underline{true})\downarrow 1)$

$Z = f_0(f_1(D''\hat{}\langle * \rangle\hat{}X',Y,Z,a,b,\underline{true})\downarrow 2)$

Now by Lemma 2: $Y = f_0(\langle a\rangle\hat{}D_1\hat{}\langle * \rangle)$

$Z = f_0(\langle b\rangle\hat{}D_2\hat{}\langle * \rangle)$

where $D_1\hat{}D_2$ is a permutation of D''.

Now we can use induction, for $|D_i| \leq |D|-2$ for i = 1,2. So we get

$Y = D_Y\hat{}\langle * \rangle$ and $Z = D_Z\hat{}\langle * \rangle$,

D_Y and D_Z are ordered,

$D_Y\hat{}D_Z$ is a permutation of D.

Again by Lemma 2:

$f_0(X) = f_1(D''\hat{}\langle * \rangle\hat{}X', Y, Z, a, b, \underline{true})\downarrow 3 =$

$= \langle\rangle \hat{} f_2(D''\hat{}\langle * \rangle\hat{}X', D_Y\hat{}\langle * \rangle, D_Z\hat{}\langle * \rangle, \underline{true})\downarrow 3 =$

$= f_3(X', D_Y\hat{}\langle * \rangle, D_Z\hat{}\langle * \rangle)\downarrow 3 =$ (by Lemma 1)

$= D''\hat{}\langle * \rangle,$

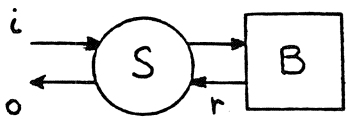
where D'' is an ordered permutation of $D_Y\hat{}D_Z$ and thus an ordered permutation of D.

2.2. Pipeline sorting

The idea is that we don't need a tree to (merge)sort a row of numbers. Treesort takes a lot of processes and after the numbers are spread out it takes log(n) steps to get the first number out of the tree.

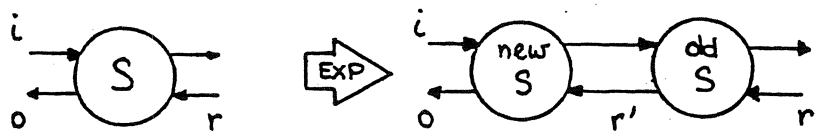
Pipeline sort takes less processes: at most n/2, where n is the number of elements to be sorted. The process expands into a linear network. Each process takes in elements as long as they can be sorted in a constant time. The sorted sequence is put out immediately after the unsorted sequence has been read in.

The program starts as follows

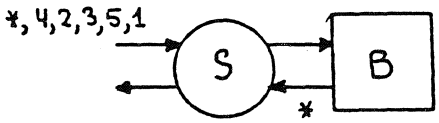


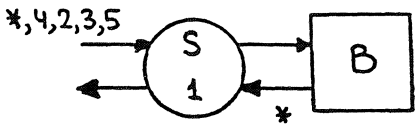
B is a bottom process: it merely sends an empty run to S. S reads numbers from i as long as it can put them into a sorted deque. If S

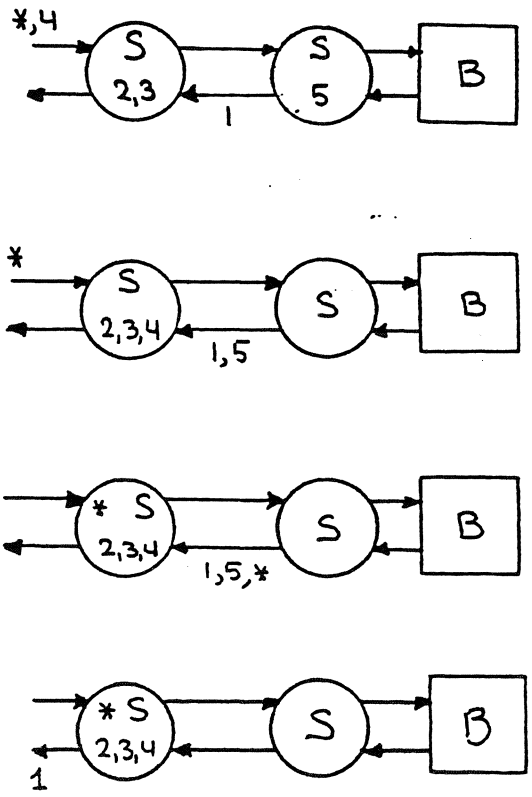
cannot do that any more it creates a new S process in front of it, merges the input from its creator (or an empty sequence from B) with its own deque and sends this run to the process it just created.



The figures below show how the file 1,5,3,2,4,* is sorted:







etc.

2.2.1. The program

```
(process start(int in u int out s):
   (int x; read(u,x);
    if x=* then write(s,*) /* nothing to sort */
           else expand
                create sort with u=u, r=rbot, s=s, empty=e, first=x
                create bottom with empty=e, o=rbot
                expand.
    if.
   );

process bottom(int in empty int out o):
   (write(o,*));
```

```
process sort(int in u,r int out s,empty int first):
   (int deque deq; insert left(deq,first);
    int next; bool expanded := false;
    repeat read(u,next);
    if next ≠ *
    then if next < left of deq then insert left(deq,next)
         elif next > right of deq then insert right(deq,next)
         else expand
              create sort with u=u, r=rs, s=s, empty=emp,first=next
              keep sort with u=emp, r=r, s=rs, empty=empty
              expand.;
         expanded := true
         if.
    if.
    until (expanded or next=*);
    /* merge */
    repeat
       read(r,next);
       if next=*
       then while deq not empty do write(s,delete left(deq))
            do.; write(s,*);
       else while deq not empty and next > left of deq do
                    write(s,delete left(deq)) do.;
            write(s,next)
       if.
    until next=*
   );

main int in unsorted;
     int out sorted;
        start(unsorted,sorted)
)
```

2.2.2. The functions

$f_0(\langle * \rangle \hat{} X) = \langle * \rangle$

$f_0(\langle a \rangle \hat{} X) = f_2(X,Y,a)\downarrow 2$
where $Y = f_1(f_2(X,Y,a)\downarrow 1)$

$f_1(X) = \langle * \rangle$

$f_2(X,Y,a) = f_3(X,Y,\langle a \rangle)$

For $deq = \langle a_1,\ldots,a_n \rangle$, $n \geq 1$, we define

$f_3(\langle a \rangle \hat{} X,Y,deq) =$ **if** $a \leq a_1$ **then** $f_3(X,Y,\langle a \rangle \hat{} deq)$
elif $a \geq a_n$ **then** $f_3(X,Y,deq \hat{} \langle a \rangle)$
else $(f_4(f_2(X,Z,a)\downarrow 1,Y,deq)\downarrow 1,\ f_2(X,Z,a)\downarrow 2)$
where $Z = f_4(f_2(X,Z,a)\downarrow 1,Y,deq)\downarrow 2$.

$f_3(\langle * \rangle \hat{} X,Y,deq) = f_4(X,Y,deq)$

$f_4(X,\langle * \rangle \hat{} Y,deq) = f_5(X,Y,deq)$

$f_4(X,\langle a \rangle \hat{} Y,deq) = f_6(X,\langle a \rangle \hat{} Y,deq)$

$f_5(X,Y,\langle\rangle) = (\langle\rangle,\langle * \rangle)$

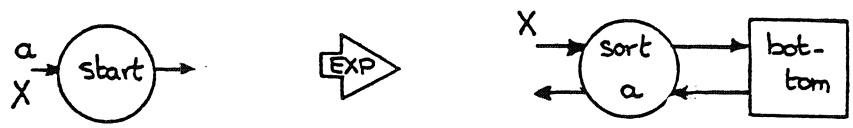
$f_5(X,Y,\langle a \rangle \hat{} deq) = (\langle\rangle,\langle a \rangle) \hat{} f_5(X,Y,deq)$

$f_6(X,\langle a \rangle \hat{} Y,\langle\rangle) = (\langle\rangle,\langle a \rangle) \hat{} f_4(X,Y,\langle\rangle)$

$f_6(X,\langle a \rangle \hat{} Y,\langle b \rangle \hat{} deq) =$ **if** $a \geq b$ **then** $(\langle\rangle,\langle b \rangle) \hat{} f_6(X,\langle a \rangle \hat{} Y,deq)$
else $(\langle\rangle,\langle a \rangle) \hat{} f_4(X,Y,\langle b \rangle \hat{} deq)$

REMARKS

- The functions f_0, f_1 and f_2 are associated with the processes start, bottom and sort respectively.
- The second line in the definition of f_0 corresponds to the expand statement in start. It can be pictured as follows:



- The second line of the defintion of f_3 corresponds to the expand statement in the process sort. It can be pictured as follows:

- deq is a double ended queue [KNUTH], i.e. a finite sequence of tokens (possibly empty), manipulated as a list.

2.2.3. The proof

Theorem. Let X = D^<*>^X', where D is finite and not containing *.
Then $f_0(X)$ = D'^<*>, where D' is an ordered permutation of D.

We prove the theorem using the following lemma's.

Lemma 1 [behaviour of f_4]. Let Y = D^<*>^Y'. Let D and deq be finite, ordered histories, not containing *. Then $f_4(X,Y,deq)$ = (<>,D'^<*>), where D' is an ordered permutation of D^deq.

Proof. By induction on |D|+|deq|. In proving the induction step one has to consider four cases: D=<>, D=<a>^D" and deq=<>, D=<a>^D" and deq=<b>^deq' with a<b, and D=<a>^D", deq=<b>^deq' with a≥b. We treat the latter case.
We have $f_4(X,Y,deq) = f_6(X,Y,deq) = (<>,<b>)^f_6(X,Y,deq')$.
Now by definition of f_4 we also have $f_4(X,Y,deq') = f_6(X,Y,deq')$, because Y has the form Y = <a>^D"^<*>^Y'. Therefore we can apply the induction hypothesis to derive the desired result.

Lemma 2 [behaviour of f_3]. Let X = D'^<*>^X', Y = D"^<*>^Y', deq ≠ <>, D',D",deq finite and not containing *, deq,D" ordered. Then $f_3(X,Y,deq)$ = (<>,D^<*>), where D is an ordered permutation of D'^D"^deq.

proof. By induction on |D'|.

The proof of the theorem can now straightforwardly be given.

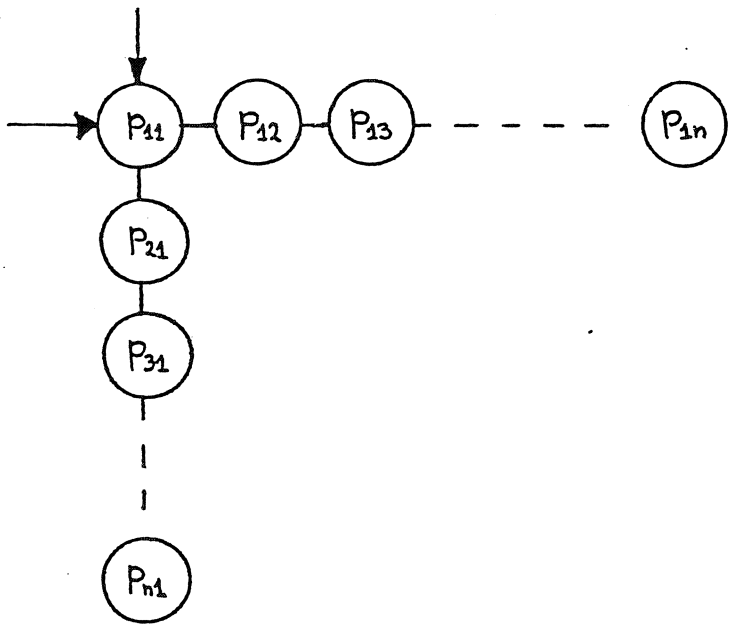
2.3. Matrix multiplication.

We present a program that multiplies square matrices. It can be adjusted for rectangular matrices, but then the program becomes more complicated, while the proof explodes.

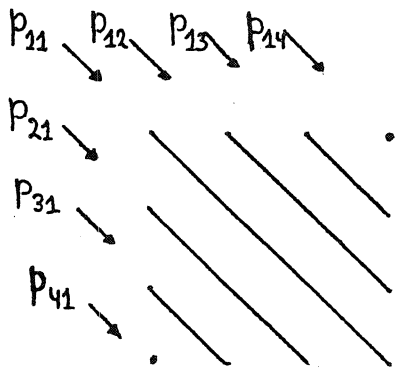
There are two input channels and one output channel. Both input channels contain a matrix, one in row format and one in column format. The program delivers the result in row format. It is also possible to deliver the result in column format, but we shall not deal with this here.

A matrix in row (column) format is a sequence of rows (columns) closed by an end-of-matrix mark (*). A row (column) is a sequence of numbers preceded by a begin-of-row (column) mark ($). The program does not check whether the matrices are square and of equal size.

If there are n rows (columns) the net will expand (stepwise) into:



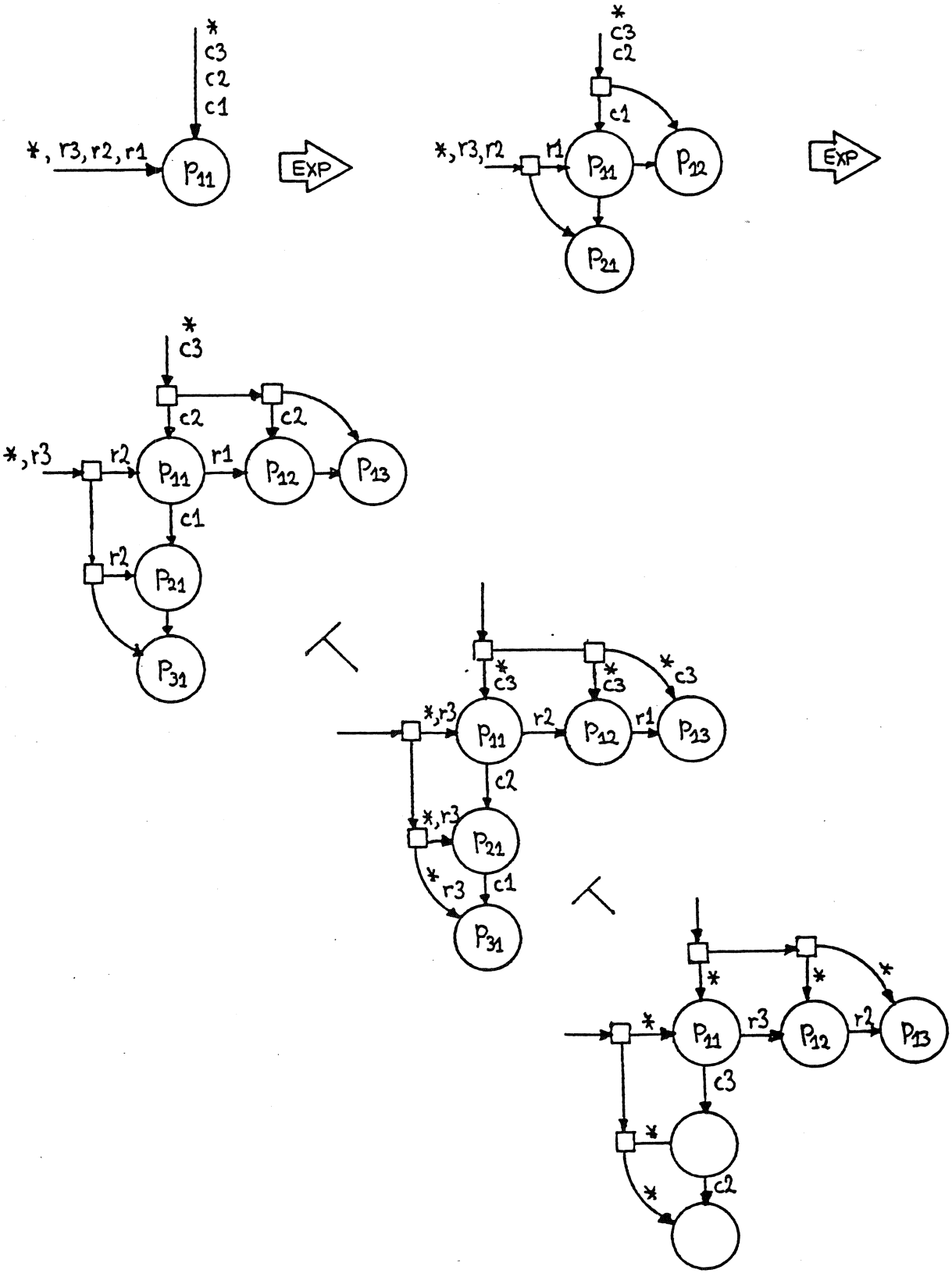
So there will be (essentially) 2n-1 processes, which are linearly interconnected. Every process p_{ij} computes a diagonal of the product matrix:



In order to compute these diagonals:

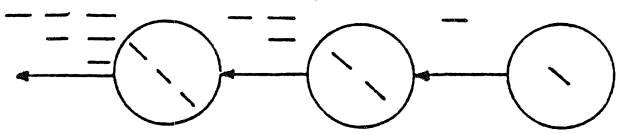
p_{11} needs $row_1,\ldots,row_n$ and
$column_1,\ldots,column_n$.
p_{12} needs $row_1,\ldots,row_{n-1}$ and
$column_2,\ldots,column_n$.
⋮
p_{1i} needs $row_1,\ldots,row_{n-i+1}$ and
$column_i,\ldots,column_n$.
p_{i1} needs $row_i,\ldots,row_n$ and
$column_1,\ldots,column_{n-i+1}$.

The figures below show the expansions for n=3. The □ processes are duplicators. A row duplicator sends the first row to the right and the rest of the matrix both to the right and down.

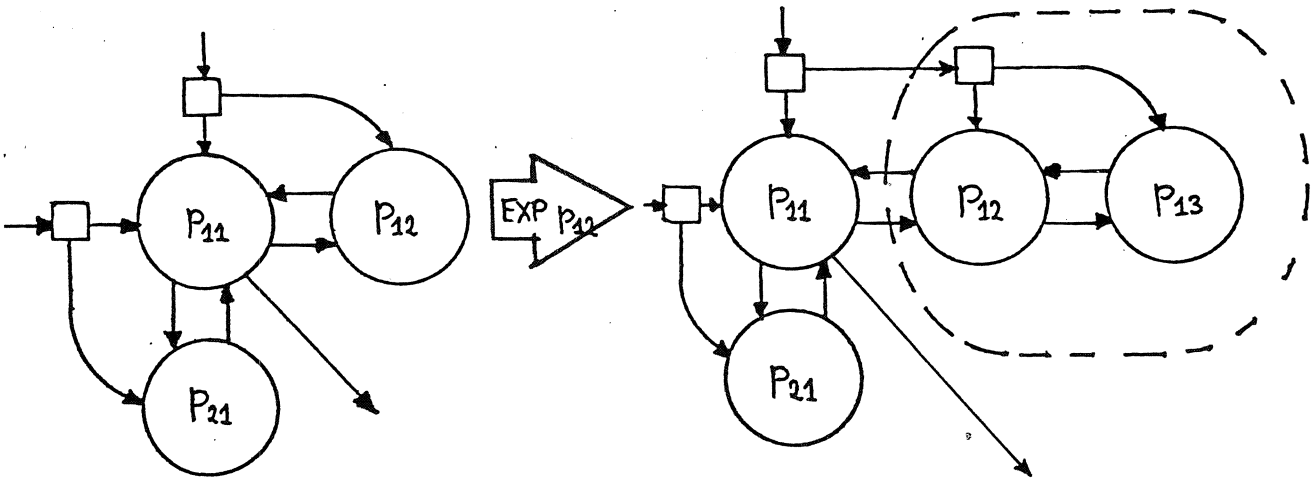

*
c3
c2
c1
*, r3, r2, r1
P11
EXP
*
c3
c2
c1
*, r3, r2
r1
P11
P12
P21
EXP
*
c3
c2
c2
*, r3
r2
P11
r1
P12
P13
c1
r2
P21
P31
*
c3
*
c3
*
c3
*, r3
P11
r2
P12
r1
P13
c2
*, r3
P21
*
r3
c1
P31
*
*
*
*
P11
r3
P12
r2
P13
c3
*
*
c2

After the diagonals have been computed, they are sent back to p_{11} such that this process can format the product matrix in row order. The process of sending back can for a row process be described as:

```
for i
   repeat send diagonal element i to the left;
          copy (part of) row_i from right to left
   until *
```

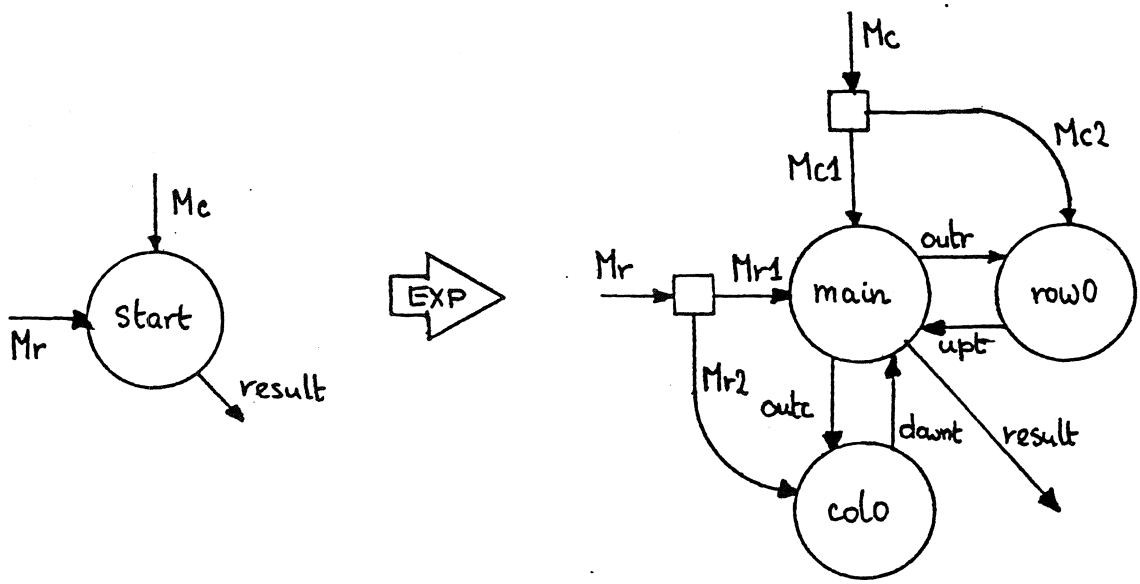


The above pictures are incomplete in that not all channels have been drawn. The picture below sketches an expansion into horizontal direction with all channels involved.



2.3.1. The program

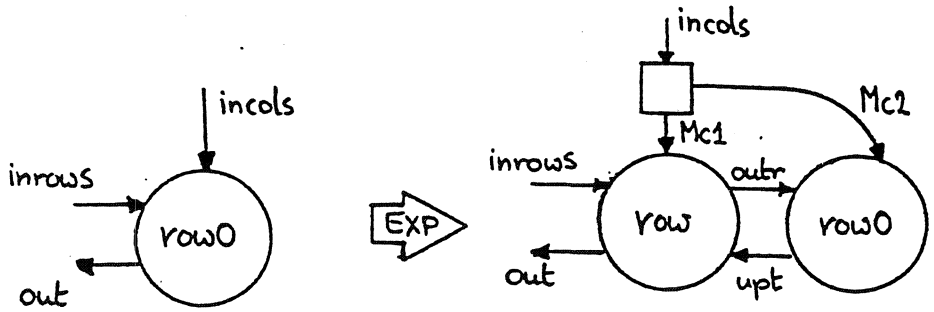
```
(process dup(int in A, int out B,C):
   (int i;
    while read(A,i); i≠* and i≠$ do write(B,i) do.;
    if i=* then write(B,*); write(C,*)
           else while i ≠* do write(B,i); write(C,i); read(A,i)
                do.;
                write(B,*); write(C,*)
    if.
   );

process start(int in Mr,Mc int out result):
   (int a,b;
    read(Mr,a); read(Mc,b);
    if a=* and b=* then write(result,*)
    else expand create dup with A=Mr, B=Mr1, C=Mr2
                create dup with A=Mc, B=Mc1, C=Mc2
                create main with Mr=Mr1, Mc=Mc1, uptriangle=upt,
                                 downtriangle=downt, outrows=outr,
                                 outcols=outc, result=result
                create row0 with incols=Mc2, inrows=outr, out=upt
                create col0 with inrows=Mr2, incols=outc, out=downt
         expand.
    if.
    );

/* This expansion can be represented graphically as follows:
```



*/

```
process main(int in Mr, Mc, uptriangle, downtriangle
             int out outrows, outcols, result):
   (queue q; /* FIFO list with operations insert, delete */
    int a,b,x;
    write(outrows,$); write(outcols,$);
    x := 0; q := empty;
    while read(Mr,a); read(Mc,b); write(outrows,a); write(outcols,b);
          a≠* and b≠*
    do if a=$ and b=$ then insert(q,x); x:=0
                      else x := x+a*b
       if.
    do.; insert(q,x);
    read(uptriangle,a);
    if a=* then write(result,$); write(result, delete(q));
                write(result,*)
       elif a=$ then
          repeat write(result,$);
             while read(downtriangle,b); b≠$ and b≠* do
                   write(result,b)
                do.;
                write(result,delete(q));
```

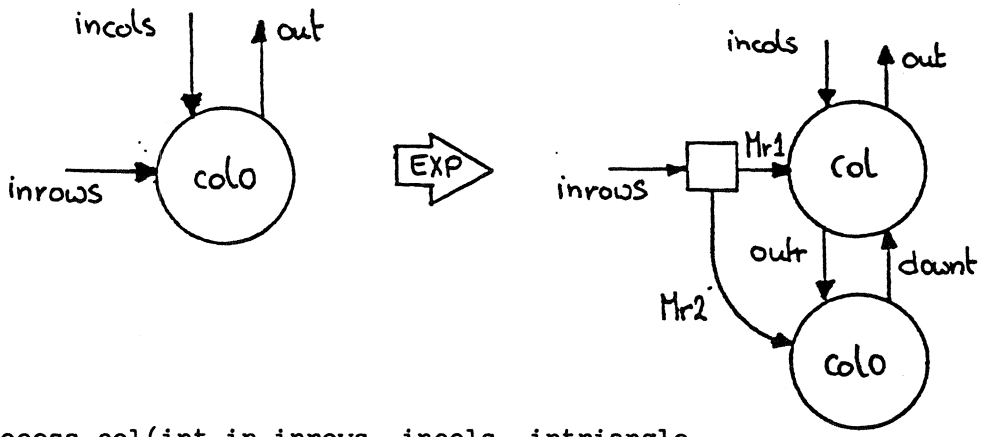
```
                if b≠* then
                   while read(uptriangle,a); a≠$ and a≠* do
                      write(result,a)
                   do.
                if.
          until b=*;
          write(result,delete(q)); write(result,*)
       if.
    if.
   );

process row0(int in incols, inrows int out out):
   (int a,b; read(incols,a);
    if a=* then write(out,*)
    else read(inrows,b);
       if a=$ and b=$ then
          expand create dup with A=incols, B=Mc1, C=Mc2
                 create row0 with incols=Mc2, inrows=outr, out=upt
                 create row with incols=Mc1, inrows=inrows,
                                 intriangle=upt, outrows=outr,
                                 outtriangle=out
          expand.
       if.
    if.
   );

/* The expansion step can be captured in the following picture:
```



```
*/

process row(int in incols, inrows, intriangle
            int out outrows, outtriangle):

   (queue q;
    int a,b,x;
    write(outrows,$); x:=0; q:=empty;
    while read(incols,a); read(inrows,b); a≠* do
       write(outrows,b);
       if a=$ and b=$ then insert(q,x); x:=0
                      else x := x+a*b
       if.
    do.;
    write(outrows,*); insert(q,x);

    while read(intriangle,a); a≠* do
       if a=$ then write(outtriangle,$); write(outtriangle,delete(q))
              else write(outtriangle,a)
       if.
    do.;
    write(outtriangle,$); write(outtriangle, delete(q));
    write(outtriangle,*)
   );

process col0(int in inrows,incols int out out):
   (int a,b; read(inrows,a);
    if a=* then write(out,*)
    else read(incols,b);
       if a=$ and b=$ then
          expand create dup with A=inrows, B=Mr1, C=Mr2
                 create col0 with inrows=Mr2, incols=outc, out=downt
                 create col with inrows=Mr1, incols=incols,
                                 intriangle=downt, outcols=outc,
                                 outtriangle=out
```

```
        expand.
      if.
    if.
   )
```

/* We have the following picture corresponding to the expansion:



*/

```
process col(int in inrows, incols, intriangle
            int out outcols, outtriangle):
   (queue q; int a,b,x;
    q := empty; x := 0; write(outcols,$);
    while read(inrows,a); read(incols,b); a≠* do
       write(outcols,b);
       if a=$ and b=$ then insert(q,x); x:=0
                      else x := x+a*b
       if.
    do.;
    write(outcols,*); insert(q,x);
    write(outtriangle,$);
    repeat read(intriangle,a);
       if a=$ or a=* then
          write(outtriangle,delete(q))
       if.; write(outtriangle,a)
    until a=*;
   );
```

main

int in matc, matr,

int out matres,

start(matr,matc,matres)

)

2.3.2. The functions

- d_0 (corresponds to process dup):

$d_0(<a>\hat{}A) = (<a>,<>)\hat{}d_0(A)$

$d_0(<*>\hat{}A) = (<*>,<*>)$

$d_0(<\$>\hat{}A) = (<\$>,<\$>)\hat{}d_1(A)$

$d_1(<a>\hat{}A) = (<a>,<a>)\hat{}d_1(A)$

$d_1(<\$>\hat{}A) = (<\$>,<\$>)\hat{}d_1(A)$

$d_1(<*>\hat{}A) = (<*>,<*>)$

- m_0 (corresponds to process start):

$m_0(<*>\hat{}A,<*>\hat{}B) = <*>$

$m_0(<\$>\hat{}A,<\$>\hat{}B) = m_1(d_0(A)\downarrow 1,d_0(B)\downarrow 1,C,D,0,<>)\downarrow 3$

where $C = r_0(d_0(B)\ 2,<\$>\hat{}m_1(d_0(A)\downarrow 1,d_0(B)\downarrow 1,C,D,0,<>)\downarrow 1)$

$D = c_0(d_0(A)\ 2,<\$>\hat{}m_1(d_0(A)\downarrow 1,d_0(B)\downarrow 1,C,D,0,<>)\downarrow 2)$

- m_1 (corresponds to process main):

$m_1(<a>\hat{}A,<b>\hat{}B,C,D,x,q) = (<a>,<b>,<>)\ \hat{}\ m_1(A,B,C,D,x+ab,q)$

$m_1(<\$>\hat{}A,<\$>\hat{}B,C,D,x,q) = (<\$>,<\$>,<>)\ \hat{}\ m_1(A,B,C,D,0,<x>\hat{}q)$

$m_1(<*>\hat{}A,<*>\hat{}B,C,D,x,q) = (<*>,<*>,<>)\ \hat{}\ m_2(A,B,C,D,<x>\hat{}q)$

$m_2(A,B,<*>\hat{}C,<*>\hat{}D,q\hat{}<x>) = (<>,<>,<\$,x,*>)$

$m_2(A,B,<\$>\hat{}C,<\$>\hat{}D,q\hat{}<x>) = (<>,<>,<\$,x>)\ \hat{}\ m_3(A,B,C,D,q)$

$m_3(A,B,<c>\hat{}C,D,q) = (<>,<>,<c>)\ \hat{}\ m_3(A,B,C,D,q)$

$m_3(A,B,<*>\hat{}C,D,q) =$

$= m_3(A,B,<\$>\hat{}C,D,q) = (<>,<>,<\$>)\ \hat{}\ m_4(A,B,C,D,q)$

$m_4(A,B,C,<d>\hat{}D,q) = (<>,<>,<d>)\ \hat{}\ m_4(A,B,C,D,q)$

$m_4(A,B,C,<\$>\hat{}D,q\hat{}<x>) = (<>,<>,<x>)\ \hat{}\ m_3(A,B,C,D,q)$

$m_4(A,B,C,<*>\hat{}D,q\hat{}<x>) = (<>,<>,<x,*>)$

- r_0 (corresponds to process row0)

$r_0(<*>\hat{}A,B) = <*>$

$r_0(<\$>\hat{}A,<\$>\hat{}B) = r_1(d_0(A)\downarrow 1,B,C,0,<>)\downarrow 2$

where $C = r_0(d_0(A)\downarrow 2,<\$>\hat{}r_1(d_0(A)\downarrow 1,B,C,0,<>)\downarrow 1)$

- r_1 (corresponds to process row)

$r_1(<a>\hat{}A,<b>\hat{}B,C,x,q) = (<b>,<>) \hat{} r_1(A,B,C,x+ab,q)$

$r_1(<\$>\hat{}A,<\$>\hat{}B,C,x,q) = (<\$>,<>) \hat{} r_1(A,B,C,0,<x>\hat{}q)$

$r_1(<*>\hat{}A,B,C,x,q) = (<*>,<>) \hat{} r_2(A,B,C,<x>\hat{}q)$

$r_2(A,B,<c>\hat{}C,q) = (<>,<c>) \hat{} r_2(A,B,C,q)$

$r_2(A,B,<\$>\hat{}C,q) = (<>,<\$,x>) \hat{} r_2(A,B,C,q)$

$r_2(A,B,<*>\hat{}C,q) = (<>,<\$,x,*>)$

- c_0 (corresponds to col_0)

$c_0(<*>\hat{}A,B) = <*>$

$c_0(<\$>\hat{}A,<\$>\hat{}B) = c_1(d_0(A)\downarrow 1,B,C,0,<>)\downarrow 2$

where $C = c_0(d_0(A)\downarrow 2,<\$>\hat{}c_1(d_0(A)\downarrow 1,B,C,0,<>)\downarrow 1)$

- c_1 (corresponds to col)

$c_1(<a>\hat{}A,<b>\hat{}B,C,x,q) = (<b>,<>) \hat{} c_1(A,B,C,ab+x,q)$

$c_1(<\$>\hat{}A,<\$>\hat{}B,C,x,q) = (<\$>,<>) \hat{} c_1(A,B,C,0,<x>\hat{}q)$

$c_1(<*>\hat{}A,B,C,x,q) = (<*>,<>) \hat{} c_2(A,B,C,<x>\hat{}q)$

$c_2(A,B,C,q) = (<>,<\$>) \hat{} c_3(A,B,C,q)$

$c_3(A,B,<c>\hat{}C,q) = (<>,<c>) \hat{} c_3(A,B,C,q)$

$c_3(A,B,<\$>\hat{}C,q\hat{}<x>) = (<>,<x,\$>) \hat{} c_3(A,B,C,q)$

$c_3(A,B,<*>\hat{}C,q\hat{}<x>) = (<>,<x,*>)$

2.3.3. The proof

As before we will state some lemma's which will then be used to prove the main theorem. But first a definition which supplies a notation that can be used to describe the data that will travel along the channels. This data will always have a prescribed format: a sequence of rows of integers (elements from a matrix), seperated and possibly preceded by a $-sign and terminated by a *-symbol.

Definition. Suppose R_i are finite histories not containing \$ or *. Then $[R_1,\ldots,R_n]$ is defined as follows:

- $\langle * \rangle$ if n=0 (that is $[]=\langle * \rangle$).
- $\langle \$ \rangle \hat{} R_1 \hat{} [R_2,\ldots,R_n]$ if $n \geq 1$.

Furthermore we define $\{R_1,\ldots,R_n] := R_1\hat{}[R_2,\ldots,R_n]$ $(n \geq 1)$.

If the above notation will be used then we will always implicitly assume that the histories involved do not contain \$ or *, and will be finite. Lemma 1 describes the behaviour of d_0. This operator copies its input to its first output channel, and it copies all but the first row (column) of the input to its second output channel.

Lemma 1. Let $A = \{A_1,\ldots,A_n]\hat{}A'$ $(n \geq 1)$. Then

$$d_0(A) = (\{A_1,\ldots,A_n],[A_2\ldots,A_n]).$$

Proof. 1. $d_0(A) = (A_1,\langle\rangle) \hat{} d_0([A_2,\ldots,A_n]\hat{}A')$ can be proved by induction on length(A_1).

2. The Lemma then follows by induction on k.

Lemma 2 describes the behaviour of r_2. This operator collects the results from its right neighbour. On this channel it receives an upper triangle of the product matrix. The operator then adds the diagonal elements which it has computed to it and outputs a bigger upper triangle.

Lemma 2. Let $k \geq 0$. $r_2(A,B,[C_1,\ldots,C_k],\langle x_{k+1},\ldots,x_1\rangle) =$

$$= (\langle\rangle,\{\langle x_1\rangle\hat{}C_1,\ldots,\langle x_k\rangle\hat{}C_k,\langle x_{k+1}\rangle]).$$

Proof. 1. $r_2(A,B,\{C_1,\ldots,C_k],q) = (\langle\rangle,C_1) \hat{} r_2(A,B,[C_2,\ldots,C_k],q)$ for $k \geq 1$. This can be proved by induction on length(C_1).

2. The lemma follows by induction on k.

The next lemma describes the behaviour of c_2. This operator acts like r_2. Notice however the difference in the format.

Lemma 3. Let $k \geq 0$. $c_2(A,B,[C_2,\ldots,C_{k+1}],\langle x_{k+1},\ldots,x_1\rangle) =$
$= (\langle\rangle,[\langle x_1\rangle,C_2\hat{}\langle x_2\rangle,\ldots,C_{k+1}\hat{}\langle x_{k+1}\rangle])$.

Proof. 1. $c_3(A,B,\{C_2,\ldots,C_k],q) = (\langle\rangle,C_2)\hat{}c_3(A,B,[C_3,\ldots,C_k],q)$ $(k \geq 2)$
by induction on length(C_2).

2. $c_3(A,B,[C_2,\ldots,C_{k+1}],\langle x_{k+1},\ldots,x_1\rangle) =$
$= (\langle\rangle,\{\langle x_1\rangle,C_2\hat{}\langle x_2\rangle,\ldots,C_{k+1}\hat{}\langle x_{k+1}\rangle])$ $(k \geq 0)$ by induction on k.

3. The lemma now follows immediately.

Lemma 4 describes the behaviour of m_2. This operator collects the upper triangle above the main diagonal from the product matrix, and the triangle below this diagonal. It then combines it with the values on the main diagonal into the product matrix in row format.

Lemma 4. Let $k \geq 1$. $m_2(A,B,[C_1,\ldots,C_k],[D_2,\ldots,D_{k+1}],\langle x_{k+1},\ldots,x_1\rangle) =$
$= (\langle\rangle,\langle\rangle,[R_1,\ldots,R_{k+1}])$, where $R_1 = \langle x_1\rangle\hat{}C_1$;
for $2 \leq i \leq k$: $R_i = D_i\hat{}\langle x_i\rangle\hat{}C_i$; $R_{k+1} = D_{k+1}\hat{}\langle x_{k+1}\rangle$.
In case k=0 we obtain $(\langle\rangle,\langle\rangle,[\langle x_1\rangle])$ as an answer.

Proof. 1. The case k=0 is immediate.

2. For $k \geq 1$ we have $m_3(A,B,\{C_1,\ldots,C_k],D,q) =$
$= (\langle\rangle,\langle\rangle,C_1)\ \hat{}\ m_3(A,B,[C_2,\ldots,C_k],D,q)$
by induction on length(C_1).

3. Similarly we have for $k \geq 1$ $m_4(A,B,C,\{D_1,\ldots,D_k],q) =$
$= (\langle\rangle,\langle\rangle,D_1)\ \hat{}\ m_4(A,B,C,[D_2,\ldots,D_k],q)$.

4. For $k \geq 1$ we have by induction on k:
$m_3(A,B,\{C_1,\ldots,C_k],\{D_2,\ldots,D_{k+1}],\langle x_{k+1},\ldots,x_2\rangle) =$
$= (\langle\rangle,\langle\rangle,\{C_1,D_2\hat{}\langle x_2\rangle\hat{}C_2,\ldots,D_k\hat{}\langle x_k\rangle\hat{}C_k,D_{k+1}\hat{}\langle x_{k+1}\rangle])$

The next lemma describes the behaviour of r_1. This operator first forms an upper diagonal, stores this in its queue q, also passes the first n rows to its right neighbour, and finally acts like r_2. The process row, corresponding to this operator, will always be started with n=m-1.

The history B′ which appears in the lemma is needed because otherwise the induction argument in lemma 6 would not work.

Lemma 5. Let $1 \leq n \leq m$, $A=\{A_1,\ldots,A_n]$, $B=\{B_1,\ldots,B_m]\hat{}B'$, $|A_i|=|B_i|$.
Let $\langle a_1,\ldots,a_k\rangle.\langle b_1,\ldots,b_k\rangle$ be the vector product $a_1b_1+\ldots+a_kb_k$.
Then $r_1(A,B,C,0,q) = (\{B_1,\ldots,B_n],\langle\rangle) \hat{} r_2(\langle\rangle,B'',C,\langle A_nB_n,\ldots,A_1B_1\rangle\hat{}q)$ for some B".

Proof. 1. $1 \leq n \leq m$ and $|A_1|=|B_1|$ implies:

$$r_1(\{A_1,\ldots,A_n],\{B_1,\ldots,B_m]\hat{}B',x,q) =$$
$$= (B_1,\langle\rangle) \hat{} r_1([A_2,\ldots,A_n],[B_2,\ldots,B_m]\hat{}B',x+A_1B_1,q).$$

This can be proved with induction on $|A_1|$.
2. The lemma then follows by induction on n.

We now get a lemma which describes the behaviour of r_0. This operator takes the last n columns of the input matrix in column form, and the first m rows from the input matrix in row format (the corresponding process will only be started with m=n+1). The operator then computes the upper triangle of the product matrix consisting of n diagonals.

Lemma 6. Let $0 \leq n \leq m$, $|A_i|=|B_i|$. Then $r_0([A_1,\ldots,A_n],[B_1,\ldots,B_m]\hat{}B') =$
$= [R_1,\ldots,R_n]$, where $R_i = \langle A_iB_i,\ldots,A_nB_i\rangle$.

Proof. By induction on n. The basic case n=0 is immediate, so assume $n \geq 1$.
Define $A = \{A_1,\ldots,A_n]$, $B = \{B_1,..,B_m]$.

$$r_0(\langle\$\rangle\hat{}A,\langle\$\rangle\hat{}B\hat{}B') = r_1(d_0(A)\downarrow 1,B\hat{}B',C,0,\langle\rangle)\downarrow 2 = \text{(lemma 1)}$$
$$= r_1(A,B\hat{}B',C,0,\langle\rangle)\downarrow 2 = (*),$$

where $C = r_0(d_0(A)\downarrow 2,\langle\$\rangle\hat{}r_1(d_0(A)\ 1,B\hat{}B',C,0,\langle\rangle)\downarrow 1) = \text{(lemma 1)}$
$= r_0([A_2,\ldots,A_n],\langle\$\rangle\hat{}r_1(A,B\hat{}B',C,0,\langle\rangle)\downarrow 1) = \text{(lemma 5)}$
$= r_0([A_2,..,A_n],\langle\$\rangle\hat{}B\hat{}r_2(\langle\rangle,B'',C,\langle A_nB_n,\ldots,A_1B_1\rangle)\downarrow 1) =$
$= r_0([A_2,\ldots,A_n],[B_1,\ldots,B_m]\hat{}X) = \text{(ind)}$
$= [R_2,\ldots,R_n]$,
where $R_i = \langle A_iB_{i-1},\ldots,A_nB_{i-1}\rangle$.

So (*) $= r_1(A,B\hat{}B',[R_2,\ldots,R_n],0,<>)\downarrow 2 =$ (lemma 5)

$= <> \hat{}\ r_2(<>,B'',[R_2,\ldots,R_n],<A_nB_n,\ldots,A_1B_1>)\ 2 =$ (lemma 2)

$= [R_1',\ldots,R_n']$, where $R_i'=<A_iB_i>\hat{}R_{i+1}=<A_iB_i,\ldots,A_nB_i>$

$R_n'=<A_nB_n>$

Lemma's 7 and 8 are the counterparts of 5 and 6 but now for c_1 and c_0.

Lemma 7. Let $1\leq n\leq m$, $A=\{A_1,\ldots,A_n]$, $B=\{B_1,\ldots,B_m]\hat{}B'$, $|A_i|=|B_i|$. Then
$c_1(A,B,C,0,q) = (\{B_1,\ldots,B_n],<>)\ \hat{}\ c_2(<>,B'',C,<A_nB_n,\ldots,A_1B_1>\hat{}q)$.
Proof. Like lemma 5.

Lemma 8. Let $0\leq n\leq m$, $|A_i|=|B_i|$. Then $c_0([A_1,\ldots,A_n],[B_1,\ldots,B_m]) =$
$= [R_1,\ldots,R_n]$, where $R_i = <A_iB_1,\ldots,A_iB_i>$.
Proof. Like lemma 6.

Lemma 9 describes the behaviour of m_1. This operator has as inputs the input matrices of the whole program. It computes the main diagonal of the product matrix in q, and then acts like m_2.

Lemma 9. Let $n\geq 1$, $A=\{A_1,\ldots,A_n]$, $B=\{B_1,\ldots,B_n]$, $|A_i|=|B_i|$. Then
$m_1(A,B,C,D,0,q) = (A,B,<>)\ \hat{}\ m_2(<>,<>,C,D,<A_nB_n,\ldots,A_1B_1>\hat{}q)$.
Proof. Like lemma 5.

Theorem [behaviour of m_0]. Let $n\geq 0$, $|A_i|=|B_i|$. Then
$m_0([A_1,\ldots,A_n],[B_1,\ldots,B_n]) = [R_1,\ldots,R_n]$,
where $R_1 = <A_iB_1,\ldots,A_iB_n>$.
Proof. The case n=0 is immediate. Now suppose $n\geq 1$. Define
$A=\{A_1,\ldots,A_n]$, $B=\{B_1,\ldots,B_n]$.
$m_0(<\$>\hat{}A,<\$>\hat{}B) = m_1(d_0(A)\downarrow 1,d_0(B)\downarrow 1,C,D,0,<>)\downarrow 3 =$ (lemma 1)
$= m_1(A,B,C,D,0,<>)\downarrow 3 =$ (*).

$C = r_0(d_0(B)\downarrow 2, \langle\$\rangle \hat{} m_1(d_0(A)\downarrow 1, d_0(B)\downarrow 1, C, D, 0, \langle\rangle)\downarrow 1) =$ (lemma 1)

$= r_0([B_2,\ldots,B_n], \langle\$\rangle \hat{} A \hat{} m_2(\langle\rangle,\langle\rangle,C,D,\langle A_nB_n,\ldots,A_1B_1\rangle)\downarrow 1) =$ (lemma 9)

$= r_0([B_2,\ldots,B_n],[A_1,\ldots,A_n] \hat{} X) =$ (lemma 6)

$= [T_1,\ldots,T_{n-1}]$,

where $T_i = \langle A_iB_{i+1},\ldots,A_iB_n\rangle$.

Quite analogously $D = [S_2,\ldots,S_n]$, where $S_i = \langle A_iB_1,\ldots,A_iB_{i-1}\rangle$.

Combining this we get (by lemma 9)

$(*) = \langle\rangle \hat{} m_2(\langle\rangle,\langle\rangle,[T_1,\ldots,T_{n-1}],[S_2,\ldots,S_n],\langle A_nB_n,\ldots,A_1B_1\rangle)\downarrow 3.$

We now apply lemma 4. There are two cases:

A. $n=1$. Then $(*) = m_2(\langle\rangle,\langle\rangle,\langle *\rangle,\langle *\rangle,\langle A_1B_1\rangle)\downarrow 3 = [\langle A_1B_1\rangle]$

B. $n \geq 2$. Then $(*) = [R_1,\ldots,R_n]$, where

$R_1 = \langle A_1B_1\rangle \hat{} T_1 = \langle A_1B_1\rangle \hat{} \langle A_1B_2,\ldots,A_1B_n\rangle = \langle A_1B_1,\ldots,A_1B_n\rangle$.

$R_i = S_i \hat{} \langle A_iB_i\rangle \hat{} T_i = \langle A_iB_1,\ldots,A_iB_{i-1}\rangle \hat{} \langle A_iB_i\rangle \hat{} \langle A_iB_{i+1},\ldots,A_iB_n\rangle$

$R_n = S_n \hat{} \langle A_nB_n\rangle = \langle A_nB_1,\ldots,A_nB_{n-1}\rangle \hat{} \langle A_nB_n\rangle$.

3. CONCLUSIONS AND FUTURE DIRECTIONS OF OUR RESEARCH

We have given two sorting algorithms and a matrix multiplication algorithm, programmed in Kahn's language. We have proven these programs correct.

At this point we can make several remarks. First of all, the proofs are long because they have to deal with many details. Secondly, they are not very appealing to the intuition because we have introduced an intermediate stage between the program and its proof, namely the functions. In the third place, we did not really prove the programs correct, but we have proved the correctness of the functions which we have derived from the programs. We claimed that the functions provided the meanings of the corresponding programs but we did not specify how one obtains functions from programs, neither did we justify our claim. In fact we have been quite loose there. Kahn [KAHN74] alludes to the method of McCarthy but he does not specify this further. It is not clear how to

apply this method, once expand statements, containing keeps creep in. What clearly is needed is a formal semantics of the language, that is we need a meaning function M which takes a program and delivers the corresponding function. At the moment we are constructing such a semantics.

The proof of the program could very well be stated more intuitively if we had a Hoare system for the language. In that case we could prove properties of the language immediately from the program text. This is to be contrasted to the two step process which we have used in this paper. Presently we are trying to find such Hoare like proof rules and axioms. Once this has been obtained we can compare proofs like the ones given here with direct proofs using Hoare rules.

Another drawback of the semantics used in this paper is that, for instance, the meaning of our paradigm "hello"-"goodbye" program from section 1.1 can only state that infinite sequences of "hello"s and "goodbye"s will travel along the channels. However, it is impossible to derive that the "hello"s and "goodbye"s are interleaved, which might be the very purpose of programs like these. In our Hoare system in development it will be possible to prove properties like these.

Due to the VLSI technology processors have become relatively cheap. It is therefore promising to let processes work together, and Kahn's language seems to be useful to state such algorithms. It is needed to find efficient algorithms which can be used in such a set up, and we will continue to work on this.

Acknowledgements.

We like to thank Ruurd Kuiper who attended many of our meetings during which the ideas presented here have been developed. In several stages in the development of the algorithms Jan van Leeuwen has been a great help. The delimiter notation do...do.,if...if. was taken from Steve Pemberton's language NEWSPEAK.

References

[KAHN74] The semantics of a simple language for parallel programming, Gilles Kahn, Information processing 1974.

[KAHN77] Coroutines and networks of parallel processes, Gilles Kahn, David B. MacQueen, Information processing 1977.

[KNUTH] The art of computer programming, vol. one: fundamental algorithms. Addison-Wesley, 1969.

[SL5D1a] An overview of the SL5 programming language, Ralph E. Griswold and David R. Hanson, Report SL51Da, Dept of Comp. Science, Univ. of Arizona, 1976.

[SL5D7a] Filters in SL5, David R. Hanson, Report SL5D7a, Dept. of Comp. Science, Univ. of Arizona.