THE DENOTATIONAL SEMANTICS OF DYNAMIC NETWORKS OF PROCESSES

Wim Böhm

Arie de Bruin

RUU-CS-82-13

Augustus 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

THE DENOTATIONAL SEMANTICS OF DYNAMIC NETWORKS OF PROCESSES

Wim Böhm
Arie de Bruin

Technical Report RUU-CS-82-13
Augustus 1982

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

THE DENOTATIONAL SEMANTICS OF DYNAMIC NETWORKS OF PROCESSES

Wim Böhm
Department of Computer Science, University of Utrecht,
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Arie de Bruin
Faculty of Economics, Erasmus University,
P.O. Box 1738, 3000 DR Rotterdam, the Netherlands

Abstract

DNP (dynamic networks of processes) is a variant of the language introduced by Kahn and MacQueen [3,4]. In the language it is possible to dynamically create new processes. We present a complete, formal denotational semantics for the language, along the lines sketched by Kahn and MacQueen.

Keywords and phrases: denotational semantics, parallellism, recursively defined processes, parallel coroutines, continuation semantics.

1. Introduction

In this paper we will define the denotational semantics of DNP (dynamic networks of processes), a language introduced by Kahn and MacQueen [3,4]. A DNP program describes a network of parallel computing stations (processes) which are interconnected by channels. Processes can only communicate via these channels. The channels are possibly infinite queues of values. Communication is asynchronous. The computing stations can "expand" into subnetworks, which will be connected to the rest of the network by the original channels. The process that caused the expansion may remain active and become part of the new subnetwork. This is called a "keep".

Kahn and MacQueen define the meaning of a DNP process as a function from input histories to output histories. A history is a possibly infinite sequence modelling the values ever transmitted through a channel. In [3] an intuitive treatment of the semantics of this kind of parallel programs

is given. However, it is not specified precisely how to obtain the meaning of a single process from its program text. Furthermore they give an informal treatment of how the meaning of a network is derived from the constituent processes and the network topology. In this paper we will give a complete formal semantics of the language.

2. Syntax

To keep the definition of the semantics short we will use a stripped version of DNP, defined by the following BNF-like syntax.

We use the following syntactic classes as primitives:

$x \in Var$	Program variables
$C \in Chvar$	Channel variables
$P \in Pvar$	Process names
$t \in Exp$	Expressions
$b \in Bexp$	Boolean expression

Expressions and boolean expressions are built up from variables, constants and operators in the usual way.

$S \in Stat$ Statements

$S ::= x:=t \mid S_1;S_2 \mid$ while b do S od $\mid$ if b then S_1 else S_2 fi $\mid$ read(x,C) $\mid$ write(t,C) $\mid$ expand E

$B \in Inst$ Instantiations

$B ::= P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m) \mid$ keep $P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)$

A channel is called an **input channel** if it occurs before the semicolon and an **output channel** if it occurs after it.

$E \in Ndef$ Network definitions

$E ::= [B_1 \| \ \ldots \ \| B_k]$

where it must be possible to partition the set of all channels into three subclasses:

Global inchan(E), viz. the channels that occur once and only once as an input channel.

Global outchan(E), viz. the channels that occur once and only once as an output channel.

Internal chan(E), viz. the channels that occur twice, once as an input channel and once as an output channel.

$T \in \mathit{Decl}$ Process declarations

$T ::= P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m) \Leftarrow$ begin S end

where .all C_i are different

.all channels occurring in a read statement in S are in $\{C_1,\ldots,C_k\}$

.all channels occurring in a write statement in S are in $\{C_{k+1},\ldots,C_m\}$

.for all expand E in S

.Global inchan(E) = $\{C_1,\ldots,C_k\}$

.Global outchan(E) = $\{C_{k+1},\ldots,C_m\}$

.E contains at most one keep of the form keep $P(C_1',\ldots,C_k';C_{k+1}',\ldots,C_m')$

$A \in \mathit{Prog}$ Programs

$A ::= \langle T_1,\ldots,T_n : P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)\rangle$

where $P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)$ and all instantiations in all T_i are well-formed with respect to $T_1,\ldots,T_n$. Here well-formedness is defined as follows. Let $T_1,\ldots,T_n$ be a sequence of process declarations and (keep) $P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)$ an instantiation B. We call B well-formed with respect to $T_1,\ldots,T_n$ iff there is a T_i in $T_1,\ldots,T_n$ of the form $P(C_1',\ldots,C_k';C_{k+1}',\ldots,C_m')$.

Remarks

An expand statement "expand E" replaces the process in which it occurs by a subnetwork of processes, connected to the rest of the graph by the channels in Global inchan(E) U Global outchan(E). The processes in the subnetwork are interconnected by the channels in Internal chan(E). The restriction imposed on the class of declarations guarantees these properties for all expand statements. If an instantiation in E is a keep, then the new process inherits the data and control environment of the original process, i.e. it will proceed with the statement following "expand E". The other instantiations are fresh copies of processes starting at the first statement with all variables initialised on the value **undefined**.

3. An example program and its associated functions

The following DNP program sorts a sequence of nonnegative numbers followed by -1. This is a simplified version of pipelinesort from [1, section 2.2.]. The program starts as in figure 3.1.

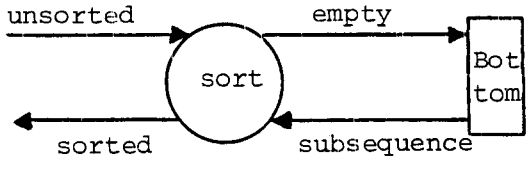

figure 3.1.

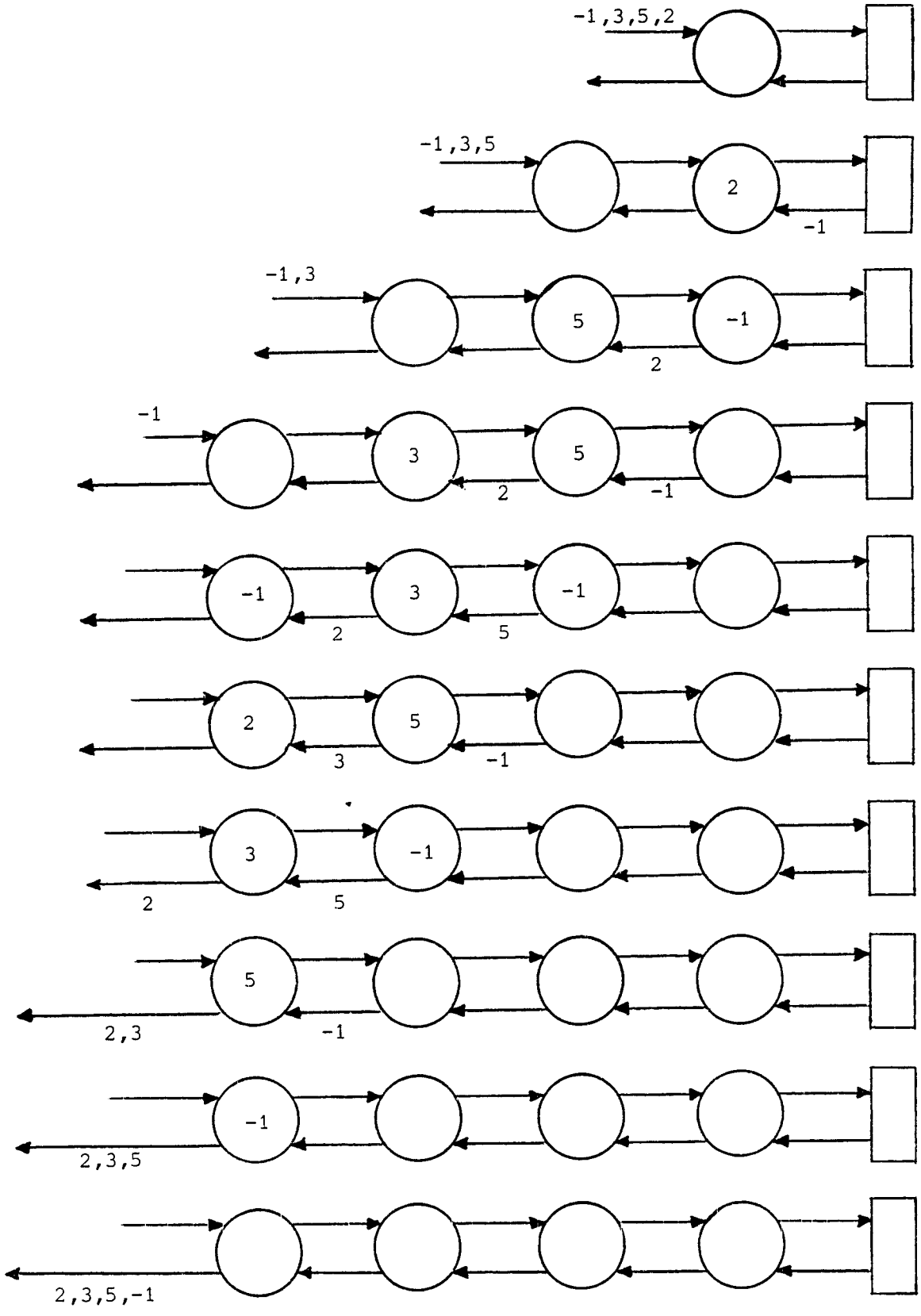

figure 3.3.

A sort process reads one number from the channel "unsorted", creates a fresh sort process in front of it, and inserts the number just read into a sorted subsequence from the channel "sorted". Sort creates a process in front of it by means of the expansion:

```
expand [sort(unsorted,subsequence1; sorted,empty1)
       ‖keep sort(empty1,subsequence; subsequence1,empty)
       ]
```

which is pictured in figure 3.2.

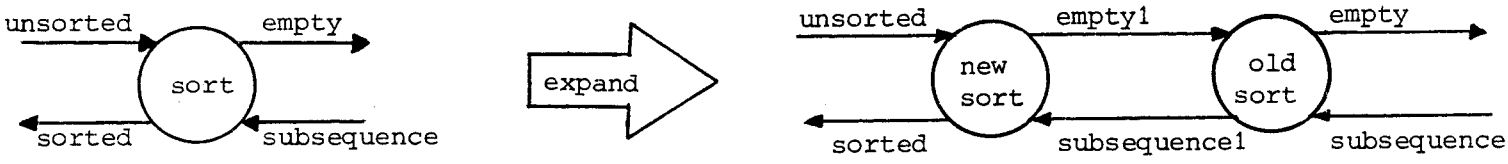

figure 3.2.

The new sort process in the picture is a fresh copy of sort; the old sort process is a keep which will manipulate the number it just read. Bottom is a process which just sends an empty (thus sorted) subsequence to the first sort process.
Sorting the sequence 1, 5, 3, -1 proceeds as shown in figure 3.3.

We now give the program text.

```
<
sort(unsorted,subsequence;sorted,empty) ⇐
begin read(x,unsorted);
   if x≧0 then expand [sort(unsorted,subsequence1;sorted,empty1)
                      ‖keep sort(empty1,subsequence;subsequence1,empty)
                      ];
      read(y,subsequence);
      while (y≧0 and y≦x)
         do write(y,sorted); read(y,subsequence) od;
      write(x,sorted)
   else read(y,subsequence)
   fi;
   while y≧0 do write(y,sorted); read(y,subsequence) od;
   write(-1,sorted)
end,
```

```
bottom(empty;subsequence) ⇐ begin write(-1,subsequence) end,

main(unsorted;sorted) ⇐
begin expand [sort(unsorted,subsequence;sorted,empty)
             ‖bottom(empty;subsequence)
             ]
end
:
main(in;out)
>
```

According to [3,4] we associate the functions from input histories to output histories f_{sort}, f_{bottom} and f_{main} with the process declarations above. These functions have the following properties

(1) $f_{bottom}(X) = \langle -1 \rangle$

(2) $f_{main}(X) = Y$,
where $\langle Y,U \rangle = f_{sort}(X,V)$
$V = f_{bottom}(U)$

(3) $f_{sort}(\langle -1 \rangle \hat{} X, \langle y \rangle \hat{} Y) = f_{copy}(y,X,Y)$

(4) $f_{sort}(\langle x \rangle \hat{} X, Y) = \langle U,V \rangle$,
where $\langle U,W \rangle = f_{sort}(X,Z)$
$\langle Z,V \rangle = f_{merge}(x,W,Y)$, for $x \geq 0$

(5) $f_{merge}(x,X,\langle y \rangle \hat{} Y) = y \geq 0 \wedge y \leq x \rightarrow (\langle y \rangle,\langle\rangle) \hat{} f_{merge}(x,X,Y),\ (\langle x \rangle,\langle\rangle) \hat{} f_{copy}(y,X,Y)$

(6) $f_{copy}(y,X,\langle z \rangle \hat{} Y) = y \geq 0 \rightarrow (\langle y \rangle,\langle\rangle) \hat{} f_{copy}(z,X,Y),\ (\langle -1 \rangle,\langle\rangle)$

The meaning of the program is the meaning of the initial network, viz. f_{main}. In these equations three forms of recursion occur. The simplest is the recursive definition of f_{merge} in (5), which stems from the fact that f_{merge} models the behaviour of a while statement. In (4) two kinds of recursion can be observed. Firstly, the histories Z and W are defined recursively, because the subnetwork in which they occur is cyclic. Secondly, f_{sort} is defined recursively, which stems from the fact that f_{sort} models the behaviour of an expand statement in sort. In the formal semantics in section 4 these complications are taken care of by the least fixed point definitions 4.2.2.4., 4.2.2.7. and 4.2.6. respectively. The above equations are sufficient to show that the network indeed yields a sorted permutation of the input sequence [1].

4. Semantics

In this section we will present the semantical domains and functions. The next section will be devoted to some explanatory remarks.

We will make use of the following notational conventions.

- If X and Y are domains then $X \to Y$ denotes the domain of all functions from X to Y. If moreover X and Y are cpo's then $[X \to Y]$ denotes the domain of all continuous functions in $X \to Y$.
- Function application associates to the left, i.e. $fabc = ((f(a))(b))(c)$.
- The $\to$ operator associates to the right, i.e. $A \to B \to C \to D = A \to (B \to (C \to D))$.
- To enhance readability, syntactical arguments are enclosed in $[\![\ ]\!]$-type brackets and continuations in $\{\}$-type brackets.
- If $f \in X \to Y$ then $f\{y/x\}$ denotes the function $\lambda x'.x'=x \to y, fx'$.
- Tuple notation: the sequence of objects $x_1,\ldots,x_n$ is denoted by $\langle x_1,\ldots,x_n\rangle$. Concatenation is denoted by $\hat{}$. Projection is denoted by subscripts, i.e. if $x = \langle a,b,c\rangle$ then $x_2 = b$.
- $\alpha \to \beta,\gamma$ denotes β if α is true and γ otherwise.

4.1. Domains

Values	$\delta \in V$ (**undefined** $\in V$)
States	$\sigma \in \Sigma = Var \to V$
Histories	$\tau \in V^\infty$
Channel contents	$\varepsilon \in Chcont = Chvar \to V^\infty$
Processes	$\alpha \in Process = [Chcont \to Chcont]$
Continuations	$\theta \in Cont = \Sigma \to Process = \Sigma \to [Chcont \to Chcont]$
Process generators	$\beta \in Prgen = Chvar^* \to Chvar^* \to Process$, with the restriction that the resulting processes write on their output channels only, i.e. for all ε, $\beta\langle c_1,\ldots,c_k\rangle\langle \bar{c}_1,\ldots,\bar{c}_n\rangle\varepsilon c = \langle\rangle$ for all $c \notin \{\bar{c}_1,\ldots,\bar{c}_n\}$.
Environments	$\gamma \in Env = (Pvar \to Prgen) \times (Pvar \to [Process \to Prgen])$,

4.2. Functions

4.2.1. $M{:}Exp \to \Sigma \to V$ and $M{:}Bexp \to \Sigma \to \{true,false\}$ are assumed to be predefined.

4.2.2. $M{:}Stat \to [Env \to [Cont \to (\Sigma \to Process)]]$

4.2.2.1. $M[\![x:=t]\!]\gamma\theta\sigma\varepsilon = \theta(\sigma\{M[\![t]\!]\sigma/x\})\varepsilon$

4.2.2.2. $M[\![s_1;s_2]\!]\gamma\theta\sigma\varepsilon = M[\![s_1]\!]\gamma\{M[\![s_2]\!]\gamma\theta\}\sigma\varepsilon$

4.2.2.3. $M[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!]\gamma\theta\sigma\varepsilon = M[\![b]\!]\sigma \to M[\![S_1]\!]\gamma\theta\sigma\varepsilon,\ M[\![S_2]\!]\gamma\theta\sigma\varepsilon$

4.2.2.4. $M[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]\gamma\theta\sigma\varepsilon = M[\![b]\!]\sigma \to M[\![S]\!]\gamma\{M[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]\gamma\theta\}\sigma\varepsilon,\ \theta\sigma\varepsilon$

4.2.2.5. $M[\![\textbf{read}(x,C)]\!]\gamma\theta\sigma\varepsilon = \varepsilon C = \langle\rangle \to \lambda C.\langle\rangle,\ \theta\sigma'\varepsilon'$,

where $\sigma' = \sigma\{\textbf{first}(\varepsilon C)/x\}$ and $\varepsilon' = \varepsilon\{\textbf{rest}(\varepsilon C)/C\}$

4.2.2.6. $M[\![\textbf{write}(t,C)]\!]\gamma\theta\sigma\varepsilon = \lambda C'.C' \equiv C \to \langle M[\![t]\!]\sigma\rangle^{\frown}\theta\sigma\varepsilon C',\ \theta\sigma\varepsilon C'$

4.2.2.7. $M[\![\textbf{expand } E]\!]\gamma\theta\sigma\varepsilon = \lambda C.C \in \textbf{Global outchan}(E) \to (\mu\Phi^*)C,\ \langle\rangle$,

where Φ^*: *Chcont* → *Chcont* is defined by

$$\Phi^*\varepsilon' = M[\![E]\!]\gamma\theta\sigma(\varepsilon'\{\varepsilon C/C\}_{C \in \textbf{Global inchan}(E)})$$

4.2.3. M: *Inst* → [*Env* → [*Cont* → (Σ → *Process*)]]

4.2.3.1. $M[\![P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)]\!]\gamma\theta\sigma = \gamma_1 P\langle C_1,\ldots,C_k\rangle\langle C_{k+1},\ldots,C_m\rangle$

4.2.3.2. $M[\![\textbf{keep } P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)]\!]\gamma\theta\sigma = \gamma_2 P(\theta\sigma)\langle C_1,\ldots,C_k\rangle\langle C_{k+1},\ldots,C_m\rangle$

4.2.4. M: *Ndef* → [*Env* → [*Cont* → (Σ → *Process*)]]

$$M[\![[B_1 \| \ldots \| B_k]]\!]\gamma\theta\sigma = \textbf{concat}(M[\![B_1]\!]\gamma\theta\sigma,\ldots,M[\![B_k]\!]\gamma\theta\sigma),$$

where $\textbf{concat}(\alpha_1,\ldots,\alpha_k)\varepsilon C = \begin{cases} \alpha_i\varepsilon C & \text{for the smallest } i \text{ such that} \\ & \alpha_i\varepsilon C \neq \langle\rangle, \text{ if there is such an } i \\ \langle\rangle & \text{otherwise} \end{cases}$

4.2.5. M: *Decl* → [*Env* → *Env*]

$$M[\![P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m) \Leftarrow \textbf{begin } S \textbf{ end}]\!] = \langle\gamma_1\{\varphi_1/P\},\gamma_2\{\varphi_2/P\}\rangle,$$

where $\varphi_2 = \lambda\alpha.\lambda\langle C'_1,\ldots,C'_s\rangle.\lambda\langle C'_{s+1},\ldots,C'_t\rangle.$

$s \neq k$ or $t \neq m \to \lambda\varepsilon.\lambda C.\langle\rangle,$

$\lambda\varepsilon.\lambda C.C \equiv C'_{s+i} \to \alpha\varepsilon' C_{s+i},\ \langle\rangle$

where $\varepsilon' = \lambda C.C \equiv C_i \to \varepsilon C'_i,\ \langle\rangle$

and $\varphi_1 = \varphi_2(M[\![S]\!]\gamma\{\lambda\sigma.\lambda\varepsilon.\lambda C.\langle\rangle\}(\lambda x.\textbf{undefined}))$

4.2.6. M: *Prog* → *Process*

$$M[\![T_1,\ldots,T_n : P(C_1,\ldots,C_k;C_{k+1},\ldots,C_m)>]\!] =$$

$\overline{\gamma}_1 P\langle C_1,\ldots,C_k\rangle\langle C_{k+1},\ldots,C_m\rangle$,

where $\overline{\gamma} = (M[\![T_1]\!]\circ\ldots\circ M[\![T_n]\!])\overline{\gamma}$

... I said: "Well, what about
the other people in the world
who might enjoy the melody of
the black page but could not
really approach its statistical
density in its basic form".
So I went to work ...

Frank Zappa,
The black page, #2

5. Discussion

In the headings of the following subsections we will refer to the corresponding semantic clauses from section 4. We assume acquaintance with the concepts of denotational semantics as provided in e.g. [2].

5.1. Domains (4.1.)

5.1.1. Values, states and histories

V denotes the set of all values that can be assumed by the program variables. One special value **undefined** is added, because we don't want to be bothered by nonessential nondeterminism caused by uninitialized variables; we initialize all variables on **undefined**. States are defined in the usual way. Each process has its own state, there is no sharing of variables between processes. A history is a finite or infinite sequence of values. On the class of histories we impose a cpo structure by defining $\tau_1 \sqsubseteq \tau_2$ iff τ_1 is a prefix of τ_2. The bottom element in V^∞ is the empty sequence <>.

5.1.2. Channel contents and Processes

In section 3 we defined (like Kahn and MacQueen) the meaning of a process as a function from tuples of histories to tuples of histories. Our approach follows these lines, but as we will define the semantics of a process declaration by induction on the structure of its body we cannot easily use functions on tuples of histories because when we define the meaning of a statement containing a channel variable, the position of that channel variable in the input or output tuple is no longer known. Instead, we apply the mechanism as used for states: the meaning of a statement is a function from channel contents to channel contents, where a channel contents associates a history with every channel variable.

A process α takes the histories on its input channels, which are assumed to be there before the computation starts and yields the histories on the output channels, consisting of all values written. We allow only continuous functions from *Chcont* to *Chcont*. Notice that there are infinite objects in

Chcont, the results of infinite computations. Usually infinite computations are modelled by a bottom element, but our semantics yields a well defined and useful result.

5.1.3. Continuations

Direct semantics does not seem appropriate. Consider the meaning of composition. This should be something like

$$M[\![S_1;S_2]\!]\sigma\varepsilon = M[\![S_2]\!](M[\![S_1]\!]\sigma\varepsilon) = \varepsilon'.$$

where σ is the initial state, ε models the contents on the input channels and ε' models the result on the output channels. Now $M[\![S_1]\!]\sigma\varepsilon$ must yield an intermediate result, and this poses at least three problems:

(i) What if S_1 blocks on trying to read from an empty channel? A special intermediate state **blocked** could be introduced, but this can hardly be called an elegant solution.

(ii) An intermediate result must at least contain an intermediate state σ', an intermediate contents of the input channels, and the output resulting from S_1. Now $M[\![S_2]\!]$ must concatenate its own output to S_1's output, but concatenation $(\lambda\tau_1.\lambda\tau_2.\tau_1\hat{}\tau_2)$ is not continuous.

(iii) What should $M[\![$expand E$]\!]$ look like?

We will use continuation semantics. We give $M[\![S]\!]$ an extra argument θ, a continuation, which is meant to model the future of the computation, i.e. θ supplies the meaning of the statements to be executed after S. In other words, if θ specifies how execution proceeds once the right hand end of S has been reached, $M[\![S]\!]\theta$ specifies execution starting from the left hand end of S. More information about continuations can be found in [2]. The domain of all continuations is *Cont*: the future of a computation is modelled by a θ which takes a state and a contents of input channels, and yields the contents of the output channels.

5.1.4. Process generators and environments

$M[\![S]\!]$ needs one more argument, an environment, to obtain the meaning of the process names occurring in S. The meaning of a process is a continuous function from *Chcont* to *Chcont*. A process declaration yields a process in terms of the formal channel names, but an instantiation must yield a process in terms of the actuals. To this end the domain of process generators is introduced. A generator accepts a finite list of actual input channels and a finite list of actual output channels and yields the actual process. The restriction imposed in the definition of *Env* is needed to

ensure continuity of **concat**$(\alpha_1,\ldots,\alpha_n)\varepsilon$ in ε (cf. definition of M⟦E⟧, 4.2.4.), where the α_i are processes generated by the instantiations occurring in a network definition E.

For normal instantiations (i.e. not keeps) the formal process and thus the corresponding process generator is derived from its declaration. This is modelled by the first component of environments. For keeps the formal process will be supplied explicitly, and this is modelled by the second component of environments.

5.2. The function M

5.2.1. Instantiations M⟦B⟧ (4.2.3.)

An instantiation is always part of an expand statement. An instantiation either creates a fresh copy of a process (normal instantiation) or resumes the process in which the expand statement occurs (keep). The meaning M⟦B⟧$\gamma\theta\sigma$ of an instantiation B in an environment γ is a process $\alpha \in$ *Process* which corresponds to executing the body of B. See also 5.1.4. The arguments θ and σ are those associated with the expand statement in which the instantiation occurs. For normal instantiations we obtain the process generator from the first component of the environment and the process name. We then apply this generator to the actual channels and this yields the actual process.

A keep corresponds to the expanding process, which remains active after execution of the expansion. This process will start executing the statements (dynamically) following the expand statement and is therefore described by the continuation θ associated with the expand statement. The starting state is the state σ in which the original process expanded. So the formal process we need is $\theta\sigma$.

5.2.2. Declarations M⟦T⟧ (4.2.5.)

The meaning M⟦T⟧γ of a declaration T in an environment γ is a new environment: with a process name P two functions φ_1 and φ_2 are associated (see also the discussion of the domain *Env* in 5.1.4.). First φ_2; this function expects a process α specified in terms of the formal input and output channels, and two lists of actual channel names. It yields the actual process. The formal-actual transformation proceeds in two stages. The contents of the actual input channels are given by ε. First ε is transformed to ε' which models the same input but now in terms of the formals. Thus $\alpha\varepsilon'$ yields the right output, but in terms of the formals. This is rewritten to an element of *Chcont* in terms of the actuals in the λ-expression

$$\lambda\varepsilon.\lambda C.C \equiv C'_{s+i} \rightarrow \alpha\varepsilon' C_{s+i}, \langle\rangle.$$

For the function φ_1 a formal process does not have to be supplied explicitly. It will be derived from the declaration T by evaluating its body with respect to the empty continuation in an inital state where all variables are undefined.

5.2.3. Programs $M[\![A]\!]$ (4.2.6.)

The meaning of a program is the meaning of its body evaluated in the environment determined by the declarations. Notice that the definition is recursive in $\overline{\gamma}$. This is needed because there can be recursive instantiations in the bodies of the T_i.

5.2.4. Statements $M[\![S]\!]$ (4.2.2.)

$M[\![S]\!]\gamma\theta\sigma\varepsilon$ yields a channel contents ε' describing the histories on the output channels resulting from executing S followed by the future computation as described by the continuation θ. S is executed in a state σ with respect to an environment γ where the contents of the input channels is given by ε.

5.2.4.1. Assignment (4.2.2.1.)

$M[\![x:=t]\!]\gamma\theta\sigma\varepsilon$ yields the contents of the output channel by first of all evaluating the assignment x:=t in σ (yielding an updated state $\sigma\{M[\![t]\!]\sigma/x\}$) and after that proceeding as given by the continuation θ.
Therefore, the effect of an assignment is captured by applying the continuation to the updated state. The contents of the input channels do not change because no input is read.

5.2.4.2. Composition (4.2.2.2.)

Composition is handled in the standard way: evaluation of $S_1;S_2$ with respect to θ is equivalent to evaluation of S_1 with respect to {evaluation of S_2 with respect to θ}, cf. 5.1.3.

5.2.4.3. Conditional (4.2.2.3.)

The result of evaluating "if b then S_1 else S_2 fi" with respect to environment γ, continuation θ, state σ and input channel ε, is either the result of evaluating S_1 with respect to these parameters (namely if b evaluated in σ yields true), or the result of evaluating S_2 (otherwise).

5.2.4.4. Repetition (4.2.2.4.)

The statement "while b do S od" is equivalent to

"if b then S; while b do S od

else skip

fi"

Evaluating the meaning of the latter statement gives us 4.2.2.4.

Notice the recursion here. Equation 4.2.2.4. is an informal way of writing down the least fixed point expression

$$M[\![\text{while b do S od}]\!]\gamma\theta = \mu[\lambda\theta'.\lambda\sigma.M[\![b]\!]\sigma \rightarrow M[\![S]\!]\gamma\theta'\sigma,\ \theta\sigma].$$

A similar remark applies to the definition of $\overline{\gamma}$ in the definition of the meaning of programs (4.2.6.).

5.2.4.5. Input (4.2.2.5.)

In defining the meaning of a read statement two cases can be discriminated. If the input channel is empty the process is blocked, it will have no effect on its output channels anymore, i.e. it yields $\lambda C.\langle\rangle$. As the process is blocked the continuation, which models the future of the computation, is ignored. Remark that our semantics assumes that all input which will be supplied to a process is given by the initial channel contents, there is no such thing modelled in our semantics as a process waiting for input. If the input channel is not empty then read(x,C) is equivalent to the assignments

x := first element of C; C := rest of C.

5.2.4.6. Output (4.2.2.6.)

Consider the write statement "write(t,C)" evaluated with respect to a continuation θ. For all channels except C this statement is equivalent to the empty statement. The output history on C consists of the value of t followed by what will be written on C in the future.

A discussion of the expand statement will be given after we have treated network definitions.

5.2.5. Network definitions M[[E]] (4.2.4.)

To model the expand statement we need to find the (smallest) solution of a set of equations in history-valued variables, derived from the topology of the new network. Consider as an example an expansion into the net in figure 5.1.

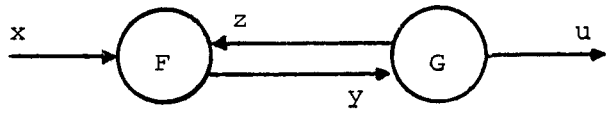

figure 5.1.

According to Kahn the global behaviour of the net is described by an operator which takes an input hsitory x and yields an output history u. This operator is derived by solving the equations:

$$y = F(x,z)$$
$$\langle z,u\rangle = G(y)$$

This is equivalent to deriving the least fixed point of $\lambda\langle y,z\rangle.\langle F(x,z),G(y)\downarrow 1\rangle$, where $G(y)\downarrow 1$ corresponds to output on the channel labelled z. In our approach we follow the same line of thought but now in terms of channel contents. This means that we need to find the least fixed point of an operator from *Chcont* to *Chcont*. This is accomplished in two stages. First we describe the behaviour of the processes in the network as if they were not interconnected, i.e. the internal channels occur twice but the two occurrences are not related yet. In terms of the example above we derive the operator M[[E]]: which is pictured in figure 5.2.

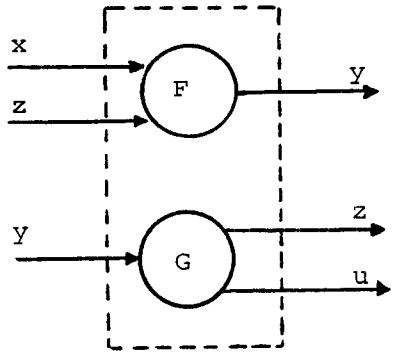

figure 5.2.

5.2.6. Expand statements $M[\![\text{expand } E]\!]$ (4.2.2.7.)

Here is the second stage. We have derived an operator $M[\![E]\!]\gamma\theta\sigma$ and now we will transform it into an operator Φ^* of which we will take the fixed point. Φ^* essentially connects the internal channels. In terms of the example, Φ^* rephrases the operator $\lambda\langle y,z\rangle.\langle F(x,z), G(y)\downarrow 1\rangle$ as a function from *Chcont* to *Chcont*. We cannot simply take the fixed point of $M[\![E]\!]\gamma\theta\sigma$ because the global input given by ε in the definition must be supplied explicitly. Notice that the results on the internal channels are invisible from outside the expand statement.

5.2.7. The existence of M

We have to show that M is well defined in the sense that for any syntactical object Δ, $M[\![\Delta]\!]$ is an element of the right domain, e.g. for every instantiation B we have that $M[\![B]\!]\gamma\theta\sigma$ must be continuous in γ and θ. This result then guarantees the existence of the fixed points occurring in the definition.

These properties can straightforwardly be shown by induction on the complexity of Δ.

References

[1] BÖHM, A.P.W. and A. DE BRUIN, Dynamic networks of parallel processes, Report IW 192/82, Amsterdam, Mathematical Centre, 1982.

[2] GORDON, M., The denotational description of programming languages, New York, Springer Verlag, 1979.

[3] KAHN, G., The semantics of a simple language for parallel programming, in: J.L. Rosenfeld (ed.), IFIP74, Amsterdam, North-Holland Publ. Comp., 1974, pp. 471-475.

[4] KAHN, G. and D.B. MacQUEEN, Coroutines and networks of parallel processes, in: B. Gilchrist (ed.), IFIP77, Amsterdam, North-Holland Publ. Comp., 1977, pp. 993-998.