A TERM MODEL FOR CCS

M.C.B. Hennessy and G.D. Plotkin

Dept. of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ
Scotland

1. Introduction

In a series of papers [Hen2, Mil1, Mil4-7] Milner and his colleagues have studied a model of parallelism in which concurrent systems communicate by sending and receiving values along lines. Communication is synchronised in that the exchange of values takes place only when the sender and receiver are both ready, and the exchange is considered as a single event; this kind of communication is also found in Hoare's model [Hoa]. In these papers, and particularly in [Mil5], a notation for expressing systems is introduced which (as remarked in [Hen2]) can be considered as a programming language, called here CCS- (Milner's) Calculus of Communicating Systems. More precisely there will be a family of languages incorporating these ideas and in this paper we study one such language.

In sections 2 and 3 we give a formal definition of the syntax of our version of CCS and then give an operational semantics by axiomatising the capabilities of programs to communicate along lines. A number of laws for behaviours were proposed in [Mil5]. A simpler version of these laws, based on a programming language not involving recursion or value passing, was justified and shown complete in [Hen2] by using an operational equivalence relation based on an operational semantics. The initial algebra for these laws then easily gave a denotational semantics for the simple language which was fully abstract with respect to the operational semantics. This meant that, for programs, being operationally equivalent was just the same as having the same denotation.

In section 4 we tentatively propose a certain operational preorder on programs; this seems more appropriate than an equivalence. In section 5 we give a formal proof system for a fragment of our language (excluding recursion but allowing value passing and nonconvergence). This system provides analogues of all the laws in [Hen2,Mil5] as well as adding some unexpected rules, and it is shown sound and complete in Theorem 5.1. In section 6 we give a variant of Milner's behaviour algebras, [Mil5], that enables us to give a denotational semantics for CCS. An initial fully abstract model is obtained in Theorem 6.7 by employing a term model construction based on the so-called behaviourally finite terms and our operational preorder (cf. [Mil3][Ber]). Available models using powerdomains (such as in [Mil1]) are not fully abstract but perhaps such a model could be obtained along the lines of [Hen1].

2. The Syntax of CCS

The syntax is parameterised on certain sets and functions as follows:

1. AVar - a given countably infinite set of arithmetic variables, ranged over by (the metavariable) x.
2. AExp - a given countably infinite set of arithmetic expressions, ranged over by e, and assumed to contain the set, N, of integers.

3. BExp - a given countable set of Boolean expressions ranged over by b, and assumed to contain the set T = {tt,ff} of truthvalues.
4. Δ - a given countable set of line names ranged over by α, β and γ.
5. $Proc_k$ (one for each integer k) - a given countably infinite set of procedure names of degree k ranged over by P_k.

In addition it is assumed that all expressions e, b have given finite sets FV(e), FV(b) of free arithmetic variables and that it is possible to substitute arithmetic expressions, e', for arithmetic variables, x, in expressions e,b to obtain expressions [e'/x]e, [e'/x]b of the same type. Finally we assume that there are relations $e =_\alpha e'$, $b =_\alpha b'$ of α-conversion in expressions and that free variables, substitution and α-conversion have the usual properties (see [Cur], [Hin1]).

Now we can give the main syntactic sets:

6. Ren - the set of renamings, ranged over by S, is the set of finite partial functions from Δ to Δ.
7. Term - the set of terms, ranged over by t, u and v, is given by the grammar:
t::= NIL | (t + u) | (t | u) | t[S] | (αx.t) | α(e,t) | (if b then t else u) |
$P^k(e_1,\ldots,e_k)$ (k ∈ N)

Free variables, substitution and α-equivalence are extended to terms in the evident way, the only new binding operators being the (αx.—).
8. Dec - the set of declarations, ranged over by d, and including all sequences of the form:
$P^1_{k(1)}(x_{11},\ldots,x_{1k(1)}) \Leftarrow t_1,\ldots,P^l_{k(l)}(x_{l1},\ldots,x_{lk(l)}) \Leftarrow t_l$
where $l \geq 0$, and $P^1_{k(1)},\ldots,P^l_{k(l)}$ are all different and include any procedure name occurring in any of the t_i, and $FV(t_i)$ is a subset of $\{x_{i1},\ldots,x_{ik(i)}\}$ for i = 1, k.
In other words we impose the usual restrictions on simultaneous recursive definitions.
9. Prog - the set of programs, ranged over by p, q and r, whose elements have the form:

letrec d in t

where any procedure name occurring in t also occurs in d and where $FV(t) = \emptyset$.

The present version of CCS differs from what is expected in the light of [Mil5] in that integers are the only type of values considered, terms and programs are not sorted on the names of the lines on which they input and on which they output, and restriction and relabelling have been replaced by the more general renaming. Notationally we have replaced (α?x:t) and (α!e:t) by the more neutral (αx.t) and α(e,t); we might also have allowed a more flexible form of procedure definition than the present two-level one. None of these variations should materially affect our results.

3. Operational Semantics

Following the intuitions in [Mil1,Mil4-7] we understand the behaviour of programs in

terms of their capabilities. These include the capability to input a value off a line or to output a value on a line or to make some internal communication. Further the recursive definitions allow computation to proceed forever without any communication occurring. For any given program none, one or several of each of these kinds of possibilities may obtain. It is presumably possible to invent an abstract machine with these capabilities, following [Lan, Weg] but we prefer the more direct, if more abstract, method of axiomatising them. This method could be called axiomatic operational semantics and has also been pursued in [Hen1,Hen2].

For the communication capabilities we need the following binary relations:

Input Here we have the relations, $p \xrightarrow{\alpha?m} q$, (one for each α in Δ and m in N) meaning that program p has the capability of inputting m off α and q represents the remaining capabilities of p after this communication.

Output Here we have the relations, $p \xrightarrow{\alpha!m} q$, (one for each α in Δ and m in N) meaning that program p can output m on α and q represents the rest of p.

Internal Communication Here we have the relation, $p \xrightarrow{\tau} q$, meaning that program p can perform an internal communication and q represents the rest of p. (We do not need any detailed knowledge about which internal communication took place as we intend to treat all the possibilities as indistinguishable.)

For the possibility of infinite computation without communication we axiomatise the property:

Convergence The property, $p \downarrow$, means that program p cannot compute forever without any communication occurring.

To understand programs it is necessary to understand expressions and terms (the latter in the context of declarations). For the first we just assume that any expressions e and b have values $[\![e]\!]$ and $[\![b]\!]$ in, respectively, N and T provided they are closed (have no free variables). For the second we introduce a little axiomatic system. Let Com be the set of communication capabilities where:

$$Com = \{\alpha?m \mid \alpha \in \Delta, m \in N\} \cup \{\alpha!m \mid \alpha \in \Delta, m \in N\} \cup \{\tau\}$$

It is ranged over by the variable c. The formulae of our axiomatic system have the form, $t \xrightarrow[d]{c} u$ or $t \downarrow d$ where t and u are closed and where d is the context of declarations in which the communication c is made or the convergence occurs. The rules have the form $F_1,\ldots,F_m \Rightarrow G_1,\ldots,G_n$ $(m \geq 0, n > 0)$ meaning that if $F_1,\ldots,F_m$ are theorems so are $G_1,\ldots,G_n$ (where the F_i and G_j are formulae). The relations that hold are to be just those whose corresponding formulae are provable.

Rules

NULLITY

1. $\Rightarrow$ (NIL $\downarrow$ d)

Although NIL cannot communicate it converges.

AMBIGUITY

1. $t \xrightarrow[d]{c} t' \Rightarrow (t+u) \xrightarrow[d]{c} t',\ (u+t) \xrightarrow[d]{c} t'$
2. $t \downarrow d, u \downarrow d \Rightarrow (t+u) \downarrow d$

The capabilities of (t+u) are those of t and u, with commitment to whichever is exercised.

COMPOSITION

1. $t \xrightarrow[d]{c} t' \Rightarrow (t|u) \xrightarrow[d]{c} (t'|u),\ (u|t) \xrightarrow[d]{c} (u|t')$
2. $t \xrightarrow[d]{\alpha!m} t',\ u \xrightarrow[d]{\alpha?m} u' \Rightarrow (t|u) \xrightarrow[d]{\tau} (t'|u'), (u|t) \xrightarrow[d]{\tau} (u'|t')$
3. $t \downarrow d,\ u \downarrow d \Rightarrow (t|u) \downarrow d$

The capabilities of (t|u) are those of t and u, but without commitment, together with an internal communication. This is handshake or synchronised communication.

RENAMING

1. $t \xrightarrow[d]{\alpha?m} t' \Rightarrow t[S] \xrightarrow[d]{S\alpha?m} t'[S]$ (if S is defined at α)
2. $t \xrightarrow[d]{\alpha!m} t' \Rightarrow t[S] \xrightarrow[d]{S\alpha!m} t'[S]$ (if S is defined at α)
3. $t \xrightarrow[d]{\tau} t' \Rightarrow t[S] \xrightarrow[d]{\tau} t'[S]$
4. $t \downarrow d \Rightarrow t[S] \downarrow d$

Renaming serves to relabel or remove communication capabilities.

INPUT

1. $\Rightarrow (\alpha x.t) \xrightarrow[d]{\alpha?m} [m/x]t,\ (\alpha x.t) \downarrow d$

The term $(\alpha x.t)$ can input any integer off α and the rest is obtained by binding x to m in t.

OUTPUT

1. $\Rightarrow \alpha(e,t) \xrightarrow[d]{\alpha!\llbracket e \rrbracket} t,\ \alpha(e,t) \downarrow d$

The term $\alpha(e,t)$ can output the value of e on α and t is the rest.

CONDITIONAL

1. $t \xrightarrow[d]{c} t' \Rightarrow (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \xrightarrow[d]{c} t'$ (if $\llbracket b \rrbracket = tt$)
2. $u \xrightarrow[d]{c} u' \Rightarrow (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \xrightarrow[d]{c} u'$ (if $\llbracket b \rrbracket = ff$)
3. $(t \downarrow d) \Rightarrow (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \downarrow d$ (if $\llbracket b \rrbracket = tt$)
4. $(u \downarrow d) \Rightarrow (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \downarrow d$ (if $\llbracket b \rrbracket = ff$)

PROCEDURE CALL

1. $[\llbracket e_1 \rrbracket/x_1] \ldots [\llbracket e_k \rrbracket/x_k]t \xrightarrow[d]{c} t' \Rightarrow P^k(e_1,\ldots,e_k) \xrightarrow[d]{c} t'$
   (if $P^k(x_1,\ldots,x_k) <= t$ is in d)
2. $[\llbracket e_1 \rrbracket/x_1] \ldots [\llbracket e_k \rrbracket/x_k]t \downarrow d \Rightarrow P^k(e_1,\ldots,e_k) \downarrow d$
   (if $P^k(x_1,\ldots,x_k) <= t$ is in d)

It is now easy to define the behaviour of programs by similar rules:

PROGRAMS

1. $t \xrightarrow[d]{c} t' \Rightarrow (\underline{\text{letrec}}\ d\ \underline{\text{in}}\ t) \xrightarrow{c} (\underline{\text{letrec}}\ d\ \underline{\text{in}}\ t')$
2. $(t \downarrow d) \Rightarrow (\underline{\text{letrec}}\ d\ \underline{\text{in}}\ t)$

As in [Hen2] we take the view that internal communication is not observable and so wish to define observable communication capabilities, $p \overset{c}{\Longrightarrow} q$, and convergence $p \Downarrow$ by the rules:

OBSERVABLE COMMUNICATION

1. $\Rightarrow (p \overset{\tau}{\Longrightarrow} p)$
2. $(p \overset{\tau}{\Longrightarrow} p'), (p' \xrightarrow{c} q'), (q' \overset{\tau}{\Longrightarrow} q) \Rightarrow (p \overset{c}{\Longrightarrow} q)$

Note that this defines $p \overset{\tau}{\Longrightarrow} q$ which, unfortunately, does not seem to correspond to anything observable.

OBSERVABLE CONVERGENCE

1. $p\downarrow,\ q\Downarrow$ (for all q such that $p \xrightarrow{\tau} q$) $\Rightarrow p\Downarrow$

This looks like an infinitary rule but it can be shown that if $p \Downarrow$ then $\{q \mid p \xrightarrow{\tau} q\}$ is finite; $p\Downarrow$ means that p cannot compute forever without an input or output communication occurring.

We will be interested in such properties of programs as are determined by the trees of communication capabilities and convergences issuing from them. The issue of fairness will be neglected; in the present context that might mean ruling out certain infinite branches of the tree as unfair. From the point of view of Petri and his followers ([Pet]) the capabilities correspond to possible events and there is a structure of concurrency and conflict on these events which we are also ignoring.

4. An Operational Preorder for Programs

An operational equivalence relation on a simpler kind, Prog', of programs than ours was introduced in [Hen2]; here we introduce an operational preorder on our kind. It should be admitted that our definition is just something that works, being based on the ideas in [Hen2] and intuitions about powerdomains [Plo,Smy] and Scott-Strachey semantics generally.

In the case of [Hen2] a simpler kind, Com', of communication capabilities was appropriate and there all programs converged. The function, E, on relations on programs was defined by putting, for any p,q in Prog':

$$p\,E(\sim)\,q \equiv (\forall c,p'.p \overset{c}{\Longrightarrow} p' \supset \exists q'.q \overset{c}{\Longrightarrow} q' \wedge p' \sim q') \wedge (\forall c,q'.q \overset{c}{\Longrightarrow} q' \supset \exists p'.p \overset{c}{\Longrightarrow} p' \wedge p' \sim q').$$

Then the operational equivalence relation, $\sim$, was defined as $\bigcap_n \sim_n$ where $\sim_0$ is the universal relation on programs and, for any n, $\sim_{n+1} = E(\sim_n)$. As it happened the communication relations, $\overset{c}{\Longrightarrow}$, obeyed the image finiteness condition that for all p, the set $\{q \mid p \overset{c}{\Longrightarrow} q\}$ is finite; this can be used to prove that $\sim$ is the maximal fixed-point of E.

All this can be understood, to some extent, in terms of an operational difference relation $\#$ on programs, which we take to be generated by the following rules:

I Symmetry $p \# q \Rightarrow q \# p$

II Communication $p \stackrel{c}{\Rightarrow} p'$, $p' \# q'$ (whenever $q \stackrel{c}{\Rightarrow} q'$) $\Rightarrow p \# q$ (for any c).

The complement of $\#$ is the maximal fixed-point of E and so, by image-finiteness, it must be $\sim$. Image-finiteness ensures the rules are finitary and so differences between programs can somehow be detected from a finite amount of information about their behaviour.

Unfortunately in our case image-finiteness fails. For example consider the program:

P_∞ = letrec I(y) <= $(\alpha x.\beta(y,NIL))$ + I(y + 1) in I(0)

Then $P_\infty \xrightarrow{\alpha ?0} \beta(m,NIL)$ for any integer m. Further we have also the possibility of nonconvergence, which suggests using a preorder, $\lesssim$, instead of an equivalence.

After some experimentation we were led to defining maps, Q^F, on preorders which employed some finiteness ideas (hence the superscript); we hope later to give a properly justified map of this kind. Now, define $p \Downarrow c$ by:

$p \Downarrow c$ iff $p \Downarrow \wedge \forall p'(p \stackrel{c}{\Rightarrow} p' \supset p' \Downarrow)$

For any finite subset, F, of Com define a function Q^F on relations over Prog by:

$$p Q^F(\lesssim) q \equiv (1).\ (\forall c \in F, p'.p \stackrel{c}{\Rightarrow} p' \supset \exists q'.q \stackrel{c}{\Rightarrow} q' \wedge p' \lesssim q') \wedge$$
$$(2).\ (\forall c \in F.p \Downarrow c \supset ((a) q \Downarrow c \wedge$$
$$(b)\ \forall q'.q \stackrel{c}{\Rightarrow} q' \supset \exists p'.p \stackrel{c}{\Rightarrow} p' \wedge p' \lesssim q'))$$

Next take $\lesssim_0^F$ to be the universal relation over Prog and for all n, put $\lesssim_{n+1}^F = Q^F(\lesssim_n^F)$ and define the operational preorder, $\lesssim_0$, by:

$p \lesssim_0 q$ iff $\forall F,n.\ p \lesssim_n^F q$

Following the ideas in [Hen2] we regard p as less than q not just when p is operationally less than q, but when p is less than q in all contexts of possible use. A term context is just a term $u[\cdot]$ with a "hole" in it (a formal definition is omitted); it can be filled in with a term t to give a term u[t] which is of use as part of a program if it is closed. For programs we are also interested in adding extra recursive definitions. We take a program context to be of the form, $v[\cdot]$ = letrec $[\cdot]$, d' in $u[\cdot]$, (where $u[\cdot]$ is a term context); it can be filled in with the program p = letrec d in t to give $v[p] =_{def}$ letrec d,d' in u[t] which is of use if it is a program (this means that u[t] must be closed and there should be no syntactic difficulties with the declarations). The contectual preorder, $\lesssim$, on programs can now be defined:

$p \lesssim q$ iff $\forall r[\cdot].(r[p] \wedge r[q]$ are programs$) \supset r[p] \lesssim_0 r[q]$

The slightly awkward notion of program context seems appropriate in the light of the two-level structure of programs.

5. A Proof System

We would have liked here to present a complete proof system characterising the operational preorder on programs; instead we consider a fragment of the language without procedure calls, but with terms for internal communication and nonconvergence. The set, BTerm, of basic terms is given by:

$$t ::= \mathrm{NIL} \mid (t+u) \mid (t|u) \mid t[S] \mid (\alpha x.t) \mid \alpha(e,t) \mid (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \mid \tau(t) \mid \Omega$$

For the operational semantics of closed basic terms we define $t \xrightarrow{c} t'$ $(c \in \mathrm{Com})$ and $t\downarrow$. The rules NIL, AMBIGUITY, COMPOSITION, RENAMING, INPUT, OUTPUT, CONDITIONAL are like those for terms, just dropping the component for declarations. The new rules are:

INTERNAL COMMUNICATION

1. $\Rightarrow \tau(t) \xrightarrow{\tau} t$ 2. $\Rightarrow \tau(t)\downarrow$

CONVERGENCE

There are no rules for Ω and so it has no communication behaviour and (unlike NIL) does not converge.

Next $\stackrel{c}{\Longrightarrow}$, $\Downarrow$ and $\sqsubseteq_0$ on closed basic terms are defined as before. Taking contexts, $v[\cdot]$, as basic terms, with a hole, a natural contextual preorder, $\sqsubseteq$ for open basic terms is:

$$t \sqsubseteq u \text{ iff } \forall v[\cdot], \forall \rho \in \mathrm{AEnv}.\ \rho(v[t]) \sqsubseteq_0 \rho(v[u])$$

using the obvious extension of arithmetic environments (note that AEnv = AVar -> N).

The next job is to characterise the operational preorder on basic terms by giving an axiomatic system for proving formulae of the forms $t \sqsubseteq u$ and $t = u$. This continues the work in [Hen2] but because of the variable binding mechanism we follow the usual pattern for λ-calculus systems rather than an algebraic style of giving universally valid equations. However the main difference from [Hen2] is that we need conditional rules, such as VIII 3, and even an infinitary rule, the ω-rule, X.

A few abbreviations will make the presentation of the rule for composition much easier. We write $\sum_{i=1,n} t_i$ for the sum, $(t_1 + (t_2 + \ldots t_n \ldots))$. A basic term is atomic iff it has one of the forms, $\alpha x.t$, $\alpha(m,t)$, $\tau(t)$ or Ω. Binary functions, $|_i$, (for i=1,3) on basic terms are defined by:

$$(t\ |_1 u) = \begin{cases} \alpha x.(t'|u) & (t = \alpha x.t') \\ \alpha(m,t'|u) & (t = \alpha(m,t')) \\ \tau(t'|u) & (t = \tau(t')) \\ \Omega & (t = \Omega) \\ \mathrm{NIL} & (\text{otherwise}) \end{cases}$$

and $|_2$ is defined symmetrically and,

$$(t\,|_3 u) \quad = \begin{cases} \tau(([m/x]t') \mid u') & (t = \alpha x.t', u = \alpha(m,u')) \\ \tau(t' \mid [m/x]u') & (t = \alpha(m,t'), u = \alpha x.u') \\ \text{NIL} & (\text{otherwise}). \end{cases}$$

Rules

I PARTIAL ORDER

1. $\Rightarrow t \sqsubseteq t$ 2. $t \sqsubseteq u,\ u \sqsubseteq v \Rightarrow t \sqsubseteq v$
3. $t \sqsubseteq u,\ u \sqsubseteq t \Rightarrow t = u$ 4. $t = u \Rightarrow t \sqsubseteq u,\ u \sqsubseteq t$

II SUBSTITUTIVITY

1. $t \sqsubseteq t' \Rightarrow t[S] \sqsubseteq t'[S], (\alpha x.t) \sqsubseteq (\alpha x.t'),\ \alpha(e,t) \sqsubseteq \alpha(e,t'),\ \tau(t) \sqsubseteq \tau(t')$
2. $t \sqsubseteq t',\ u \sqsubseteq u' \Rightarrow (t+u) \sqsubseteq (t'+u'), (t|u) \sqsubseteq (t'|u'), (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) \sqsubseteq (\underline{\text{if}}\ b\ \underline{\text{then}}\ t'\ \underline{\text{else}}\ u')$
3. $\Rightarrow \alpha(e,t) = \alpha(m,t)$ (if $FV(e) = \emptyset$ and $[\![e]\!] = m$)
4. $\Rightarrow (\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) = (\underline{\text{if}}\ tr\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u)$ (if $FV(b) = \emptyset$ and $[\![b]\!] = tr$)

III AMBIGUITY

1. $\Rightarrow (t + u) + v = t + (u + v)$ 2. $\Rightarrow (t + u) = (u + t)$
3. $\Rightarrow (t + t) = t$ 4. $\Rightarrow (t + \text{NIL}) = t$

IV COMPOSITION

1. $\Rightarrow (\sum_i t_i) \mid (\sum_j u_j) = \sum_i t_i\,|_1 (\sum_j u_j) + \sum_j (\sum_i t_i)\,|_2 u_j + (\sum_i \sum_j t_i\,|_3 u_j)$
(if all the t_i and u_j are atomic)

V RENAMING

1. $\Rightarrow \text{NIL}[S] = \text{NIL},\ (t + u)[S] = (t[S] + u[S]),\ \tau(t)[S] = \tau(t[S]), \Omega[S] = \Omega$
2. $\Rightarrow (\alpha x.t)[S] = (\beta x.t[S]),\ \alpha(e,t)[S] = \beta(e,t[S])$ (if $S(\alpha) = \beta$)
3. $\Rightarrow (\alpha x.t)[S] = \text{NIL},\ \alpha(e,t)[S] = \text{NIL}$ (if S is not defined at α)

VI α-CONVERSION

1. $\Rightarrow \alpha x.t = \alpha y.[y/x]t$ (if $y \notin FV(t)$)

VII CONDITIONAL

1. $\Rightarrow (\underline{\text{if}}\ tt\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) = t,\ (\underline{\text{if}}\ ff\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) = u$

VIII COMMITMENT

1. $\Rightarrow \tau(\tau(t)) = \tau(t),\ t + \tau(t) = \tau(t)$ 2. $\Rightarrow \tau(t + \Omega) = t + \Omega$
3. $\sum_i \tau(t_i) \sqsubseteq \sum_j \tau(u_j) \Rightarrow \sum_i (\alpha x.t_i) \sqsubseteq \sum_j (\alpha x.u_j)$
4. $\sum_i \tau(t_i) \sqsubseteq \sum_j \tau(u_j) \Rightarrow \sum_i \alpha(m,t_i) \sqsubseteq \sum_j \alpha(m,u_j)$

IX OMEGA

1. $\Rightarrow \Omega \sqsubseteq t.$

X ω-RULE

1. $[m/x]t \sqsubseteq [m/x]u$ (all m in N) $\Rightarrow t \sqsubseteq u$

Allowing for the differences in the nature of the proof systems, we have or can derive analogues of all the laws in [Hen2], [Mil4]. An example due to B. Mayoh shows a typical use of the ω-rule in conjunction with VIII 3. To show:

$$\alpha x.t + \alpha x.u \sqsubseteq \alpha x.\ (\underline{\text{if}}\ x{=}0\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) + \alpha x.\ (\underline{\text{if}}\ x{=}0\ \underline{\text{then}}\ u\ \underline{\text{else}}\ t)$$

it is enough by VIII 3 to show:

$\tau t + \tau u \sqsubseteq \tau(\underline{\text{if}}\ x=0\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) + \tau(\underline{\text{if}}\ x=0\ \underline{\text{then}}\ u\ \underline{\text{else}}\ t)$

and by the ω-rule, X, it is enough to show for every m that:

$\tau t + \tau u \sqsubseteq \tau(\underline{\text{if}}\ m=0\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u) + \tau(\underline{\text{if}}\ m=0\ \underline{\text{then}}\ u\ \underline{\text{else}}\ t)$

and that will follow by I, II, VII and III 2 (if $m \neq 0$).

Clearly this proof system is far from providing a complete, practical proof system for programs. It may be possible to make a complete one for programs by adding suitable formulae for programs and handling recursion by a Scott fixed-point rule (like the rules FIXP, INDUCT in [Mil2]). Perhaps one could then eliminate the ω-rule to obtain a finitary relatively complete system (see [Apt]) which could be a first step towards a practical one. In the meantime we have the following completeness theorem showing we have at least characterised $\sqsubseteq$:

Theorem 5.1 (Completeness) For all basic terms t,u: $t \sqsubseteq u$ iff $t \sqsubseteq u$ is a theorem of the above axiomatic system.

Hopefully, this result will increase the reader's confidence in our definition of the preorder, $\sqsubseteq$.

6. Natural Interpretations

When constructing term models it clarifies matters if it is known in what sense they are models and so we look for a class (even category) of possible interpretations for CCS. A reasonable choice would be some class of algebras, but the binding operators, $(\alpha x.\_\_)$, present difficulties and we do not follow a strictly algebraic treatment - at least in the usual narrow sense of algebra. What we do is treat the binding operators in a natural way: if terms have type A then for each α we use a function of type $(N \to A) \to A$ to interpret $\alpha x.t$. Note that this does not amount to the same thing as an algebraic treatment along the lines, say, of the treatment of SAL in [ADJ] - that would result in a wider class of semantics. Much attention has been paid to this kind of problem in the case of the λ-calculus ([Bar], [Hin2], [Obt]).

Our natural interpretations of CCS are ω-continuous algebras with some extra structure; see [ADJ] for a definition of strict ω-complete partial orders (called ω-cpos here), ω-continuous functions, ω-continuous algebras and strict ω-continuous homomorphisms.

Definition 6.1 The one-sorted signature, Σ, has one operator symbol, NIL, of arity 0 and one, [S], of arity one, for each S, and two, + and |, of arity two.

Definition 6.2 A natural interpretation, $\mathcal{A}$, of CCS is an ω-continuous Σ-algebra, A, together with functions:

$In_{\alpha,\mathcal{A}}$: $(N \to A) \to A$ (one for each α in Δ)

$Out_{\alpha,\mathcal{A}}$: $(N \times A) \to A$ (one for each α in Δ)

where $In_{\alpha,\mathcal{A}}$ is ω-continuous (taking $N \to A$ as the ω-cpo of all functions from N to A under the pointwise ordering) and where $Out_{\alpha,\mathcal{A}}$ is continuous in its second argument.

Definition 6.3 A homomorphism $h: \mathfrak{A} \to \mathfrak{A}'$ of natural interpretations is a strict ω-continuous homomorphism of the underlying Σ-algebras, A and A', such that for all α in Δ :

1. $\forall f \in (N \to A).h(In_{\alpha,\mathfrak{A}}(f)) = In_{\alpha,\mathfrak{A}'}(h \circ f)$
2. $\forall m \in N, a \in A.h(Out_{\alpha,\mathfrak{A}}(m,a)) = Out_{\alpha,\mathfrak{A}'}(m,h(a))$

Natural interpretations are closely related to Milner's behaviour algebras [Mil5]. The differences are:

1. Behaviour algebras allow other value sets than the integers.
2. Certain differences regarding renaming.
3. Natural interpretations have an order structure.
4. Behaviour algebras are many-sorted (on finite subsets of port labels).

The first two differences are trivial, reflecting what we have already discussed: the difference between our definition of CCS and what might be expected from [Mil5]. The third difference is needed for the treatment of recursion (or any linguistic device permitting infinite behaviours). We do not understand the significance of the last difference.

Any natural interpretation, $\mathfrak{A}$, of CCS gives rise to a denotational semantics for CCS. There are two kinds of environments, arithmetic and procedural, ranged over by ρ and π , and given by:

$AEnv = (AVar \to N)$ $\quad PEnv = \prod_{k\in\omega} Proc_k \to (N^k \to A)$ respectively. Updating environments and applying environments to expressions is defined as usual; we write K_0 for the trivial arithmetic environment $\lambda m \in N.0$.

The denotational semantics of CCS is now given by three functions all of which are also called $\mathfrak{A}$:

$\mathfrak{A}$: Term $\to$ (AEnv $\times$ PEnv $\to$ A), $\mathfrak{A}$: Dec $\to$ PEnv, $\mathfrak{A}$: Prog $\to$ A.

For the denotation of terms we just give enough equations to make the rest of the definition obvious:

3. $\mathfrak{A}[\![t|u]\!](\rho,\pi) = \mathfrak{A}[\![t]\!](\rho,\pi) \|_{\mathfrak{A}} \mathfrak{A}[\![u]\!](\rho,\pi)$
4. $\mathfrak{A}[\![\alpha x.t]\!](\rho,\pi) = In_{\alpha,\mathfrak{A}}(\lambda m \in N.\ \mathfrak{A}[\![t]\!](\rho[m/x],\pi))$
5. $\mathfrak{A}[\![\alpha(e,t)]\!](\rho,\pi) = Out_{\alpha,\mathfrak{A}}([\![\rho(e)]\!], \mathfrak{A}[\![t]\!](\rho,\pi))$
6. $$\mathfrak{A}[\![b \to t,u]\!](\rho,\pi) = \begin{cases} \mathfrak{A}[\![t]\!](\rho,\pi) & ([\![\rho(b)]\!] = tt) \\ \mathfrak{A}[\![u]\!](\rho,\pi) & ([\![\rho(b)]\!] = ff) \end{cases}$$
7. $\mathfrak{A}[\![P_k(e_1,\ldots,e_k)]\!](\rho,\pi) = \pi_k[\![P_k]\!]([\![\rho(e_1)]\!],\ldots,[\![\rho(e_k)]\!])$

It is easy to show $\mathfrak{A}[\![t]\!](\rho,\pi)$ is ω-continuous in π .

Given a procedure environment, π , any term t defines a function of its free variables and so if $FV(t) \subseteq \{x_1,\ldots\ldots,x_k\}$ we define:

$$\mathcal{F}[\![t;x_1,\ldots,x_k]\!](\pi) = \lambda m_1,\ldots,m_k \in N.\ \mathcal{A}[\![t]\!](K_0[m_1/x_1]\ldots[m_k/x_k],\pi)$$

Now the denotation of declarations is defined by

$$\mathcal{A}[\![P^1_{k(1)}(x_{11},\ldots,x_{1k(1)}) <= t_1,\ldots,P^l_{k(l)}(x_{l1},\ldots,x_{lk(l)}) <= t_l)]\!]$$
$$= Y(\lambda\pi.[\mathcal{F}[\![t_1;\ x_{11},\ldots,x_{1k(1)}]\!](\pi)/P^1_{k1}]\ldots[\mathcal{F}[\![t_l;\ x_{l1},\ldots,x_{lk(l)}]\!](\pi)/P^l_{k(l)}]\pi)$$

Finally the denotation of whole programs is given by:

$$\mathcal{A}[\![\underline{letrec}\ d\ \underline{in}\ t]\!] = \mathcal{A}[\![t]\!](K_0, \mathcal{A}[\![d]\!])$$

Definition 6.4 A natural interpretation, $\mathcal{A}$, of CCS is a model for CCS iff for all programs p,q:

$$p \mathrel{\underline{\sim}} q \Rightarrow \mathcal{A}[\![p]\!] \sqsubseteq \mathcal{A}[\![q]\!];$$

a model, $\mathcal{A}$, for CCS is fully abstract iff for all programs p,q:

$$\mathcal{A}[\![p]\!] \sqsubseteq \mathcal{A}[\![q]\!] \Rightarrow p \mathrel{\underline{\sim}} q.$$

Finally, we turn to the construction of a fully abstract term model, $\mathcal{M}$. The idea is to base $\mathcal{M}$ on the completion of a preorder, $\langle F, \leq \rangle$ of basic terms. The terms in F will all represent finite behaviours and the preorder will just be $\mathrel{\underline{\sim}}$.

Definition 6.5 The b-finite (behaviourally finite) terms are the least set, F, of closed basic terms, t, such that:
If $\{\langle c,t'\rangle \mid t \stackrel{c}{\Rightarrow} t' \not\equiv \Omega\}$ is finite and also t' is in F whenever $t \stackrel{c}{\Rightarrow} t'$, for any c, then t is in F.

For example, for any m, α x.(if x $\leq$ m then NIL else Ω) is b-finite but (αx.NIL) is not. As we have already remarked, F is a preorder under $\mathrel{\underline{\sim}}$ (restricted to F). Note that the least element of F is Ω. Further NIL is b-finite and so are t[S], (t + u) and (t|u) if t and u are; therefore we can turn F into a Σ-algebra if we define:

$$NIL_F = NIL;\quad [S]_F(t) = t[S];\quad t +_F u = (t + u);\quad t|_F u = (t|u)$$

and indeed F is even a preordered Σ-algebra in that all the operations are monotonic.

Now M, the underlying continuous Σ-algebra of $\mathcal{M}$, is taken to be the completion by directed ideals of F. Note the natural monotonic $[\cdot]$: F -> M, where $[u] = \{t \mid t \leq u\}$. The operations on M are defined in the evident element-wise way so that, for example,

$$X +_M Y = \bigcup\{[u +_F v] \mid u \in X,\ v \in Y\}$$

(By the way, we have used the countability of F in that if F were not countable the correct definition of M would use countably generated directed ideals.)

It remains to define $In_{\alpha,\mathcal{M}}$ and $Out_{\alpha,\mathcal{M}}$. For this purpose we assume from now on that for all m there is a condition, x = m, with one free variable x and the obvious meaning (i.e. $[\![[n/x](x = m)]\!] = tt$ iff n equals m). Now just define for any f: N -> M and X $\in$ M:

$$In_{\alpha,\mathcal{M}}(f) = \bigcup\{[\alpha x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ t^{(0)}\ \underline{\text{else}}\ \ldots\ \underline{\text{else}}\ (\underline{\text{if}}\ x = m\ \underline{\text{then}}\ t^{(m)}\ \underline{\text{else}}\ \Omega)\ \ldots\ ]\,|\,m \geq 0,\ t^{(i)} \in f(i)\ (\text{for } i = 0,m)\}$$

$$Out_{\alpha,\mathcal{M}}(m,X) = \bigcup\{[\alpha(m,t)]\,|\,t \in X\}$$

The closure properties of F ensure these definitions are correct and it is easy to show that $In_{\alpha,\mathcal{M}}$ and $Out_{\alpha,\mathcal{M}}$ have the required continuity properties, making $\mathcal{M}$ a natural interpretation.

Note that the definition of $\mathcal{M}$ does not use the characterisation of $\underset{\sim}{E}$ presented in section 5. However we do use it to prove:

Theorem 6.7 (Full Abstraction) The natural interpretation, $\mathcal{M}$, is a fully abstract model of CCS and, indeed, is initial in the category of models of CCS and their homomorphisms.

It is possible to construct other fully abstract models, showing that the second part of this theorem has point; it may also be the case that $\mathcal{M}$ is initial in a wider class of models including "unnatural" interpretations.

Acknowledgements

This research was carried out with the aid of SRC grant GR/A/75125. We thank R. Milner, K. Apt and the referees for their helpful suggestions.

References

[ADJ] (Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B.) (1977) Initial algebra semantics and continuous algebras. JACM, Vol. 24, No. 1, 68-95.

[Apt] Apt, K.R. (1979) Ten years of Hoare's logic, a survey. To appear.

[Bar] Barendregt, H. (1977) The type-free lambda calculus. Handbook of Mathematical Logic (ed. J. Barwise), pp. 1092-1131, Amsterdam: North Holland.

[Ber] Berry, G. (1979) Modèles Complètement Adéquats et Stables des Lambda-Calculus Typés. Thèse de doctorat d'état. Université Paris VII.

[Cur] Curry, H.B., Feys, R. and Craig, W. (1968) Combinatory Logic, Volume 1. Amsterdam: North Holland.

[Hen1] Hennessy, M.C.B. and Plotkin, G.D. (1979) Full abstraction for a simple parallel programming language. MFCS '79, Olomouc. Springer-Verlag Lecture Notes in Computer Science, Vol. 74, pp. 108-120.

[Hen2] Hennessy, M.C.B. and Milner, R. (1980) On observing nondeterminism and concurrency. ICALP '80, Noordwijkerhout. Springer Verlag Lecture Notes in Computer Science. To appear.

[Hin1] Hindley, J.R., Lercher, B. and Seldin, J.P. (1972) Introduction to combinatory logic. Cambridge: Cambridge University Press.

[Hin2] Hindley, J.R. and Longo, B. (1978) Lambda calculus models and extensionality. Universita degli Studi di Pisa, Istituto di Scienze dell'Informazione. Note Scientifiche S-78-4.

[Lan] Landin, P.J. (1966) A lambda-calculus approach. Chapter 5. Advances in Programming and Non-Numerical Computation. Pergamon Press.

[Mil1] Milne, G.J. and Milner, R. (1979) Concurrent processes and their syntax. JACM, Vol. 26, No. 2, 302-321.

[Mil2] Milner, R. (1976) Models of LCF. Mathematical Centre Tracts, 82, pp. 49-63. Amsterdam.

[Mil3] Milner, R. (1977) Fully abstract models of typed λ-calculi. Theoretical Computer Science, Vol. 4, 1-22.

[Mil4] Milner, R. (1978) Algebras for communicating systems. Proc. AFCET/SMF Joint Colloquium in Applied Mathematics (Palaiseau, France). Also available as CSR-25-78, Computer Science Department, University of Edinburgh, 1978.

[Mil5] Milner, R. (1978) Synthesis of communicating behaviour. Proc. 7th MFCS Conference, Zakopane, Poland. Springer Verlag Lecture Notes in Computer Science, Vol.64, pp. 71-83. Berlin: Springer Verlag.

[Mil6] Milner, R. (1979) An algebraic theory for synchronisation. Theoretical Computer Science 4th GI Conference, Aachen. Springer Verlag Lecture Notes Computer Science, Vol. 67, pp. 27-35. Berlin: Springer Verlag.

[Mil7] Milner, R. (1979) Flowgraphs and flow algebras. JACM, Vol. 26, No. 4, 794-818.

[Obt] Obtulowicz, A. (1977) Functorial semantics of the type free calculus. Fundamentals of Computation Theory. Springer Verlag Lecture Notes in Computer Science, Vol. 56, pp. 302-207. Berlin: Springer Verlag.

[Pet] Petri, C.A. (1976) Nichtsequentielle Prozesse. Arbeitsberichte des IMMD, Bd. 9, Heft. 8, p.57 ff. Universität Erlangen-Nurnberg. Also Non Sequential Processes, Translation by P. Krause and J. Low. Internal Report GMD-ISF-77-05, Bonn (1977).

[Plo] Plotkin, G.D. (1976) A powerdomain construction. SIAM Journal on Computing, Vol. 5, No. 3, 452-487.

[Smy] Smyth, M. (1978) Powerdomains. JCSS, Vol. 16, No. 1.

[Weg] Wegner, P. (1972) The Vienna definition language. Computing Surveys, Vol. 4, No. 1.

[Hoa] Hoare, C.A.R. (1978) Communicating sequential processes. CACM, Vol. 21, No. 8, 666-677.