THE LAWS OF OCCAM PROGRAMMING

by

A.W. Roscoe

and

C.A.R. Hoare

Technical Monograph PRG-53

February 1986

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

CONTENTS

		Page
0.	Introduction	1
1.	The laws of occam:	6
	- Laws of IF	8
	- Laws of ALT	9
	- Laws of assignment	12
	- Laws of SEQ	13
	- Laws of PAR	14
	- Laws of declaration	18
	- Laws of $\perp$	20
2.	A pre-normal form:	23
	- Syntactic approximation	35
	- Proving additional laws	39
3.	The normal form	46
	- Rule of substitution for expressions	59
	- Three more laws	61
4.	Conclusions and prospects	69
	- Deciding the equivalence of programs	70
	- Improving efficiency	73
	- Transformation to a restricted syntax	76
Appendix: A summary of the laws of occam		78
References		85

The laws of occam programming

A.W. Roscoe and C.A.R. Hoare

Abstract One of the attractive features of occam is the large number of memorable algebraic laws which exist relating programs. We investigate these laws and, by discovering a normal form for WHILE-free programs, show that they completely characterise the language's semantics.

0. Introduction

Occam [1] is a language for concurrent systems, especially those implemented on networks of communicating processors (transputers). It has been designed with simplicity and elegance as major goals. One way in which this elegance manifests itself is in the large number of algebraic laws which exist between occam programs. The aim of this paper is to investigate the set of laws and to show how they completely characterise the semantics of the language.

For simplicity we concentrate on a subset of occam: timing, priority, vectors, constants, replicators and named processes (procedures) are omitted. Our version of occam thus contains only the essential core needed to write simple programs. We expect that our work can readily be extended to versions of occam containing these features. The laws given in this paper will carry over (with occasional modification) to larger versions of the language. For theoretical reasons we will also add a few features to the language: multiple assignment, output guards in alternatives and a divergent (racing) process. In other respects we will follow the syntax and conventions introduced in [R], in particular those regarding the parallel operator. (When writing a parallel construct the programmer must declare which global variables and channels are to be assigned to each component process.)

A finite occam program is one which is WHILE-free. It may, however, contain the racing or diverging process ⊥ (equivalent to WHILE true SKIP). Much of this paper is concerned with the analysis of finite programs. This is because the absence of WHILE-loops allows proof by induction. This restriction does not lose us any power, however, because every occam program can be identified with the set of its finite syntactic approximations (a term which is defined precisely in the second section).

The first section lists the majority of the laws we require. We see how each of the laws arises out of our informal understanding of how occam constructors work. We see how algebraic laws allow us to give a precise and succinct description of each operator. The laws given are all congruences in the denotational semantics for occam reported in [R].

The second section shows how the laws introduced in the first section can transform every finite program to a form whose only constructs are IF, ALT, multiple assignment and ⊥ (the diverging process). Particular attention is paid to regularising the use of free and bound variables. We see how this work, together with continuity assumptions, allows us to prove non-trivial laws additional to those of the first section.

Even in this restricted form it is possible to write essentially different programs which are nevertheless semantically equivalent. The third section identifies a number of situations where such equivalences can arise, and develops a normal form for finite programs. Two normal form programs are semantically equivalent if and only if they are syntactically equivalent in a simple way. By showing how every finite program can be transformed to normal form we have thus produced a decision procedure for the equivalence of arbitrary finite programs. An infinitary rule based on syntactic approximation extends this to general programs. This proves that our set of algebraic laws (together with the infinitary rule and substitution) is <u>complete</u> with respect to the given denotational semantics. The algebraic laws thus yield an <u>algebraic semantics</u> for occam that is isomorphic to our chosen denotational semantics.

Finally we review the relative merits of algebraic, denotational and other forms of semantics, and in particular discuss possible applications of the algebraic laws as transformation rules.

All the laws presented in this paper are summarised in an appendix.

Even though the work in this paper is cast in terms of a specific denotational semantics, most of the laws quoted must be true in any reasonable abstract semantics for occam. We indicate several places where modifications may be required for alternative underlying semantics.

The work reported in this paper owes much to the similar work for an abstract version of CSP (i.e. with no internal state) reported in [8].

Notation

Throughout this paper we will observe the following conventions within program terms

P,Q	program fragments (processes)
C	conditional
G	guarded process
g,h,k	guards
e,f	general expressions
b	boolean expression
U	parallel declaration
x,y,z	identifiers representing variables
c,d	identifiers representing channels

Lists of identifiers and expressions are denoted $\underset{\sim}{x}$, $\underset{\sim}{e}$ respectively. $\underset{\sim}{x} + \underset{\sim}{y}$ denotes the concatenation of the lists $\underset{\sim}{x}$ and $\underset{\sim}{y}$. Occam syntax is usually linearised as in [R], and we frequently use such abbreviations as

$$\mathrm{IF}_{i=1}^{4}\ b_i\ P_i \quad (= \mathrm{IF}\ (b_1P_1,\ b_2P_2,\ b_3P_3,\ b_4P_4)).$$

Possibly empty lists of processes, conditionals and guarded processes are respectively written $\underset{\sim}{P}$, $\underset{\sim}{C}$ and $\underset{\sim}{G}$. The most general form of an ALT construct is thus ALT($\underset{\sim}{G}$).

Free and bound variables

If P is some occam term and x is a variable, we say that an occurrence of x in P is free if it is not in the scope of any declaration (other than a parallel declaration) of x in P, and bound otherwise. (These notions can easily be defined formally.) Note that x may occur both free and bound in P.

free (P)	denotes the set of all variables appearing free in P
bound (P)	denotes the set of all variables appearing bound in P

[Similar notions of free and bound occurrences can be defined for channels.]

Substitution

If x and y are variables, then $P\left[{}^{x}/_{y}\right]$ denotes the result of substituting x for every free occurrence of y in P. If x is bound at any point in P where there is a free y, systematic renaming of P's bound variables is carried out.

We similarly use the notations

$$f\left[{}^{e}/_{x}\right],\quad f\left[{}^{\underline{e}}/_{\underline{x}}\right],\quad \underline{f}\left[{}^{e}/_{x}\right] \text{ and } \underline{f}\left[{}^{\underline{e}}/_{\underline{x}}\right]$$

to denote the substitution of (lists of) expressions for (equal length lists of) variables in (lists of) expressions. Note that in general

$$e\left[{}^{\langle f_1,\ \ldots,\ f_n\rangle}/_{\langle x_1,\ \ldots,\ x_n\rangle}\right] \text{ is distinct from}$$

$$e\left[{}^{f_1}/_{x_1}\right]\cdots\left[{}^{f_n}/_{x_n}\right].$$

1. The laws of occam

In this section we visit each occam construct in turn, and uncover the laws governing it. The set of laws given is not exhaustive; we restrict ourselves to the laws needed to translate finite programs to normal form. Other laws can be deduced from these laws, either by elementary manipulation, or by structural induction on normal forms. The laws we present here provide a clear description of the semantics of each construct.

Before detailing the laws, we must decide exactly what we mean by the term "law". All our laws have the form P = Q (P,Q both being expressions representing processes). Informally this must mean that P "is essentially the same as" Q, in that, to an observer who cannot detect their internal structure, the behaviours of P and Q are indistinguishable. Further, since we will want to use our laws to transform subcomponents of compound programs, P = Q must imply that C[P] is essentially the same as C[Q] for all contexts C[.] (programs with a slot in which to place a program segment). Since we may wish to use our laws to transform an inefficient program to an observationally equivalent efficient one, our notion of equivalence will be independent of the times at which events occur. Thus P = Q does not imply that P and Q run at the same speed. Neither, for similar reasons, does it mean that P and Q require the same amount of store.

Having established the broad principles above, we hope that most of the laws will seem "clearly true". Nevertheless, it is helpful to have some underlying semantics by which to judge the laws. In our case this is provided by the denotational semantics for occam reported in [R]. All the laws we quote are congruences of that semantics in the context (described there) of environments with unbounded sets of free locations and channels. However, all laws must be interpreted as conditional upon both sides being <u>correct</u> occam, in the sense that neither side contains a syntax error. We will assume that the evaluation of every occam expression yields a value (even though it may contain division by zero or an uninitialised identifier). Thus no syntactically correct program in our restricted version of occam can contain an execution error. If the language were extended to include vectors the situation would be more difficult, and some of our laws would have to include exception conditions.

There are two limitations on the completely free use of our laws in transforming occam. The first is that, with a few of our laws, it is possible to transform a correct program C[P] (C[.] being a context) to an incorrect one C[Q]. This is usually brought about by violating the separation rules for PAR. The laws that can have this effect are marked (*), and have been set out so that only <u>right to left</u> use can bring about this difficulty. These laws may thus only be used right to left in contexts where syntactic correctness is preserved. The second limitation is that it is only occam <u>processes</u> that may be transformed: the laws do not apply to guarded processes or conditionals, even when they have the same syntax as processes. For example, the transformation of

ALT (c?x SKIP, ALT(SKIP ALT(d?x SKIP))) to

ALT (c?x SKIP, ALT(d?x SKIP))

is invalid, even though, as a process, ALT SKIP P may be transformed to P.

Each law is given a name suggestive of its use, and a number.

1. Laws of IF

The IF constructor is used to select the behaviour of a program, depending on the values of its variables. For this reason it will play a vital role in our later construction of a normal form.

IF takes as its arguments a number of conditionals. A conditional is either a (boolean) expression and a process (b P) or an IF construct. The first law permits us to unnest IFs, so that all arguments are of the first type.

(1.1) $\mathrm{IF}(\underset{\sim}{C}_1, \mathrm{IF}(\underset{\sim}{C}_2), \underset{\sim}{C}_3) = \mathrm{IF}(\underset{\sim}{C}_1, \underset{\sim}{C}_2, \underset{\sim}{C}_3)$ <IF assoc>

This is not an associative law in the usual binary sense of $a \ast (b \ast c) = (a \ast b) \ast c$, but is analogous in the context of occam's constructors, which can take an arbitrary finite number of arguments.

The second law expresses the fact that in the process $\underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_i\ P_i$, it is the first (i.e. lowest index) boolean guard to be true that activates the corresponding P_i. Thus P_i only runs if b_i is true and each of $b_1 \ldots b_{i-1}$ is false.

(1.2) $\underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_i\ P_i = \underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_i^{\ast}\ P_i$, where $b_i^{\ast} = \neg b_1 \wedge \ldots \wedge \neg b_{i-1} \wedge b_i$

<IF priority>

If the boolean guards in $\underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_i\ P_i$ are pairwise disjoint, then the order of composition is immaterial. (This is a symmetry law.)

(1.3) $\underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_i\ P_i = \underset{i=1}{\overset{n}{\mathrm{IF}}}\, b_{\pi(i)}\ P_{\pi(i)}$ for any permutation π of $\{1 \ldots n\}$ provided $b_i \wedge b_j \equiv$ false whenever $i \neq j$

<IF sym>

If two booleans guard the same process, they can be amalgamated.

(1.4) $\text{IF}(b_1\ P,\ b_2\ P,\ \underset{\sim}{C}) = \text{IF}(b_1 \vee b_2\ P,\ \underset{\sim}{C})$ <IF − ∨ distrib>

A <u>false</u> guard is never activated, and so can be discarded.

(1.5)† $\text{IF}(\underline{\text{false}}\ P,\ \underset{\sim}{C}) = \text{IF}(\underset{\sim}{C})$ <IF − <u>false</u> unit>

If none of the booleans in IF is true, the process behaves like STOP (i.e. it comes to a complete halt without terminating; a process sequentially composed with it is <u>not</u> allowed to start). Thus final clauses of conditionals which are STOP may freely be added or deleted.

(1.6)† $\text{IF}(\underset{\sim}{C},\ b\ \text{STOP}) = \text{IF}(\underset{\sim}{C})$ <IF − STOP unit>

If one branch of an IF construct is always executed, then the construct may be replaced by that branch.

(1.7) $\text{IF}(\underline{\text{true}}\ P) = P$ <IF − <u>true</u> unit>

The final IF law lets us deal with IF constructs which are nested as processes rather than as conditionals.

(1.8) $\text{IF}(\underset{\sim}{C},\ b\ \underset{i=1}{\overset{m}{\text{IF}}}\ b_i\ P_i) = \text{IF}(\underset{\sim}{C},\ \underset{i=1}{\overset{m}{\text{IF}}}\ b \wedge b_i\ P_i)$ <∧ − IF distrib>

This law will, of course, be used in combination with <IF − assoc>, which completes the unnesting.

2. <u>Laws of ALT</u>

The ALT constructor allows a process to offer a choice of possible communication options to its environment. The ALT constructor takes as arguments a number of guarded processes. A guarded process is either a guard and a process (g P) or an ALT construct. As with IF, there is a law which allows us to "unnest" ALTs.

(2.1) $ALT(ALT(\underset{\sim}{G}_1), \underset{\sim}{G}_2) = ALT(\underset{\sim}{G}_1, \underset{\sim}{G}_2)$ <ALT assoc>

This law does not have quite such a general form es that for IF (1.1). However the general form of the law can be deduced from 2.1 and the fact that ALT is fully symmetrical (see below).

The order of arguments in an ALT is immaterial.

(2.2) $\underset{i=1}{\overset{n}{ALT}}\, G_i = \underset{i=1}{\overset{n}{ALT}}\, G_{\pi(i)}$ π any permutation of $\{1 \dots n\}$

<ALT-sym>

The alternative composition of no arguments is STOP (the non-terminating process which does nothing).

(2.3) ALT( ) = STOP <ALT-STOP unit>

This law is termed a "unit" lew because, together with 2.1 and 2.2, it says that STOP is essentially the unit of ALT.

Guards may be simple (SKIP, c?x, c!e) or have a boolean component. ALTs with guards with boolean components may be reduced to IF combinations of ALTs with simple guerds by the law

(2.4) $ALT(b \& g\ P, \underset{\sim}{G}) = IF(b\ ALT(g\ P, \underset{\sim}{G}), \neg b\ ALT(\underset{\sim}{G}))$ <boolean guard elim>

In other words, a guard with a boolean component may be executed if and only if the boolean is true.

A SKIP guard is always ready, and its execution hae no effect other than to start the process which it guards. This expleins the law

(2.5) ALT(SKIP P) = P <ALT-SKIP identify>

A communication guard, on the other hand, is executed only when the process at the other end of the given channel is also willing. The effect is exactly like the corresponding single communication atomic processes

(2.6) ALT(c?x SKIP) = c?x <input>

(2.7) ALT(c!e SKTP) = c!e <output>

If an alternative is already present in an ALT, adding it again has no effect, since the set of alternatives available does not change.

(2.8) ALT(g P, $\underset{\sim}{G}$) = ALT(g P, g P, $\underset{\sim}{G}$) <ALT idempotence>

In any execution of an ALT construct, it ia the first guard to becoma ready which ia executed. If more than one guard becomes ready at the same time, the choice of which one to execute is nondeterministic (there is no left-to-right precedence rule as with IF). We can deduce from this thet if a guard g is used to guard two different processes, then whenever that guard becomes ready either copy may be activated, the choice being invisible to the environment. The two guarded processes can thus be replaced with a single one, where the process is one which nondeterministically chooses between the original pair.

(2.9) ALT(g P, g Q, $\underset{\sim}{G}$) = ALT(g ALT(SKIP P, SKIP Q), $\underset{\sim}{G}$) <guard distrib>

The laws abova do not quite catch the full range of equivalences related to ALT with SKIP guards. Three more laws reflecting fairly subtle aquivalences will be introduced in section 3, when they are required, and can be better motiveted.

We need a law for relating IF and ALT. It is a very simple law, which merely observes that the value of a boolean is unchanged by the execution of a guard that does not input to a variable appearing in the boolean.

(2.10) $$\text{IF } b \underset{i=1}{\overset{n}{\text{ALT}}} g_i\ P_i \;=\; \text{IF } b \underset{i=1}{\overset{n}{\text{ALT}}} g_i\ (\text{IF } b\ P_i)$$

provided no variable appearing in b is input in any g_i <IF - ALT distrib>

Perhaps surprisingly, this law is the only one we will need relating IF and ALT. An example of how it can be used to derive an apparently more powerful law can be found at the end of section 2.

3. <u>Laws of assignment</u>

An occam process may assign values to its variables. The atomic assignment process in occam is x:=e, which evaluates the expression e, assigns the result to the location denoted by x, and then terminates. As described in the introduction, we allow <u>multiple</u> assignments, of the form $\underset{\sim}{x}:=\underset{\sim}{e}$ where $\underset{\sim}{x}$ is a list of distinct variables, and $\underset{\sim}{e}$ is an equal-length list of expressions. The components of $\underset{\sim}{e}$ are evaluated, the results are then all assigned to the locations represented by $\underset{\sim}{x}$, and the process then terminates. The empty multiple assignment terminates without changing the state.

(3.1) $$\langle\rangle := \langle\rangle \;=\; \text{SKIP}$$ <SKIP>

The order in which the expression/variable pairs appear is of no consequence.

(3.2) $$\langle x_i \mid i = 1 \ldots n\rangle := \langle e_i \mid i = 1 \ldots n\rangle$$
$$= \langle x_{\pi(i)} \mid i = 1 \ldots n\rangle := \langle e_{\pi(i)} \mid i = 1 \ldots n\rangle$$

for π any permutation of $\{1 \ldots n\}$ <assignment sym>

The assignment of a variable's own value to itself has no effect.

(3.3)* $\underset{\sim}{x} + \underset{\sim}{y} := \underset{\sim}{e} + \underset{\sim}{y} \ = \ \underset{\sim}{x} := \underset{\sim}{e}$ <identity assignment>

There will be several laws later on which show how assignment interacts with the various constructs of the language.

4. Laws of SEQ

The SEQ constructor runs a number of processes in sequence. If it has no arguments it simply terminates.

(4.1) SEQ( ) = SKIP <SEQ – SKIP unit>

Otherwise it runs its first argument until that terminates and then runs the rest in sequence.

(4.2) SEQ (P, $\underset{\sim}{P}$) = SEQ (P, SEQ($\underset{\sim}{P}$)) <SEQ assoc>

It is possible to use 4.1 and 4.2 to transform all occurrences of SEQ within a program to binary applications, and in our transformation to normal form we will always do this. Thus the remainder of our laws for SEQ are cast in binary form.

When P does not terminate immediately, SEQ(P,Q)'s initial behaviour is just that of P. Thus SEQ distributes over both IF and ALT in its left argument.

(4.3)* $\text{SEQ}\ (\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ P_i,\ Q) \ = \ \overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ \text{SEQ}(P_i,\ Q)$ <SEQ – IF distrib>

(4.4)* $\text{SEQ}\ (\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i,\ Q) \ = \ \overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ \text{SEQ}(P_i,\ Q)$ <SEQ – ALT distrib>

On the other hand, when P does terminate immediately, SEQ(P,Q) behaves like Q modified to take account of any assignment by P.

Thus the compound operator SEQ ($\underset{\sim}{x}:=\underset{\sim}{e}$, .) can be distributed over both IF and ALT in a limited way.

(4.5)* $$SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ \overset{n}{\underset{i=1}{IF}}\ b_i\ P_i) = \overset{n}{\underset{i=1}{IF}}\ b_i \left[\underset{\sim}{e}/\underset{\sim}{x}\right] SEQ(\underset{\sim}{x}:=\underset{\sim}{e}, P_i)$$ <assignment – IF distrib>

(4.6)* $$SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ \overset{n}{\underset{i=1}{ALT}}\ g_i\ P_i) = \overset{n}{\underset{i=1}{ALT}}\ g_i \left[\underset{\sim}{e}/\underset{\sim}{x}\right] SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ P_i)$$

provided no variable which occurs in $\underset{\sim}{x}$ or $\underset{\sim}{e}$ is <u>input</u> in any g_i. <assignment – ALT distrib>

The sequential composition of two assignments to the same list of variables is easily combined to a single assignment.

(4.7)* $$SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ \underset{\sim}{x}:=\underset{\sim}{f}) = \underset{\sim}{x}:=\underset{\sim}{f}\left[\underset{\sim}{e}/\underset{\sim}{x}\right]$$ <combine assignments>

The sequential composition of a pair of assignments to different lists of variables may be reduced to a single assignment using this law with 3.2 and 3.3.

5. Laws of PAR

The occam parallel operator takes a number of processes as arguments, and runs them concurrently, with the possibility of communication between them. Communication is the only way two parallel processes can affect one another, so one parallel process cannot access a variable that another one can modify. No channel may be input from nor output to by more than one of the processes. In this paper (as in [R]) we insist that each parallel process <u>declares</u> which global variables it wishes to be able to modify, and which global channels it wishes to be allowed to input from, output to, or use privately. In the earlier paper this permitted the syntactic determination of the environment in which each component process should run. In this paper there is an additional reason: it would be unfortunate from the point of view of algebraic laws if the channel and variable alphabets of parallel processes were determined purely from the syntax of the component processes. Many of the most useful transformations (e.g. the expansion rules below) would become invalid, because on changing the syntax of the components of PAR, alphabets might be significantly altered. (For example, by commuting a communication through a PAR using 5.6 or 5.7, one might apparently remove it from the alphabet of the corresponding process.)

The syntax of these "parallel declarations" is unimportant; a suitable one may be found in [R].

A PAR command terminates as soon as all its components have. Thus the empty PAR terminates immediately.

(5.1) $\quad PAR(\ ) = SKIP$ <PAR – SKIP unit>

PAR is an associative operator, provided suitable provisions are made for alphabets.

(5.2) $$\mathop{PAR}_{i=1}^{n} U_i:P_i = PAR(U_1:P_1,\ U^{\dagger}:(\mathop{PAR}_{i=2}^{n} U_i:P_i)) \quad (n>0)$$

where $U^{\dagger}$ is the <u>union</u> of $U_2 \dots U_n$; <PAR assoc>

($U^{\dagger}$ claims all variables and private channels claimed by the U_i, claims as input (output) channels all channels occurring only as inputs (outputs) among the U_i, and claims as private channels all channels occurring both as an input and as an output among the U_i.)

As with SEQ, we will always use 5.1 and 5.2 to reduce PAR to a binary operator when transforming to normal form. Thus the rest of the laws deal only with that case. Firstly, PAR is symmetric, because the order in which processes are combined in parallel is immaterial.

(5.3) $\quad \mathrm{PAR}(U_1{:}P_1,\ U_2{:}P_2) = \mathrm{PAR}(U_2{:}P_2,\ U_1{:}P_1)$ $\qquad \langle$PAR sym$\rangle$

If one of a pair of parallel processes is a conditional, then the choice represented by that conditional may be performed before the parallel construct is entered, provided the choices are exhaustive (so that the conditional cannot stop the PAR being entered).

(5.4)† $\quad \mathrm{PAR}(U_1{:}\underset{i=1}{\overset{n}{\mathrm{IF}}}\ b_i\ P_i,\ U_2{:}Q) = \underset{i=1}{\overset{n}{\mathrm{IF}}}\ b_i\ \mathrm{PAR}(U_1{:}P_i,\ U_2{:}Q)$

provided $b_1 \vee \ldots \vee b_n \equiv \underline{\text{true}}$ $\qquad \langle$PAR – IF distrib$\rangle$

If two multiple assignments are combined in parallel, then the effect is that of a single multiple assignment. (Note that the conditions on use of variables within PAR mean that the variables of $\underset{\sim}{x}$ below do not occur in $\underset{\sim}{y}:=\underset{\sim}{f}$, nor those of $\underset{\sim}{y}$ in $\underset{\sim}{x}:=\underset{\sim}{e}$)

(5.5)† $\quad \mathrm{PAR}(U_1{:}\underset{\sim}{x}:=\underset{\sim}{e},\ U_2{:}\underset{\sim}{y}:=\underset{\sim}{f}) = \underset{\sim}{x} + \underset{\sim}{y} := \underset{\sim}{e} + \underset{\sim}{f}$ $\qquad \langle$PAR assignments$\rangle$

If a non-terminated process is put in parallel with a terminated one, then only the non-terminated one can proceed. It can perform any action other than a communication with the terminated process (which clearly cannot agree to any communication). In this context an assignment may be considered "terminated", because it cannot affect or be affected by the other process, and is free to terminate at any time.

(5.6)† $\quad$ If each g_i has one of the forms c?x, c!e or SKIP,

then $\mathrm{PAR}(U_1{:}\underset{i=1}{\overset{n}{\mathrm{ALT}}}\ g_i\ P_i,\ U_2{:}\underset{\sim}{x}:=\underset{\sim}{e}) = \underset{i\in X}{\mathrm{ALT}}\ g_i\ \mathrm{PAR}(U_1{:}P_i,\ U_2{:}\underset{\sim}{x}:=\underset{\sim}{e})$

where X is the set of indices $i \in \{1, 2, \ldots, n\}$ such that

$g_i = \mathrm{SKIP}$

or $\quad g_i = c!e \quad$ and $\quad c \in \mathrm{outs}(U_1) - \mathrm{ins}(U_2)$

or $\quad g_i = c?x \quad$ and $\quad c \in \mathrm{ins}(U_1) - \mathrm{outs}(U_2)$.

$\langle$ expansion 1 $\rangle$

(ins(U) and outs(U) are respectively the seta of input and output channels declared in U.)

If two non-terminated processes are put in parallel with one another then they can proceed independently on all actions except those which represent communication between them. If they agree on a communication, this can occur as an internal (automatic) action. This explains the following law for expanding two ALT constructs in parallel.

(5.7)† If $P = \underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\ P_i$, and $Q = \underset{j=1}{\overset{m}{\text{ALT}}}\ h_j\ Q_j$, where each g_i, h_j has one of the forms c?x, c!e or SKIP, then $\text{PAR}(U_1{:}P,\ U_2{:}Q) = \underset{r=1}{\overset{N}{\text{ALT}}}\ k_r\ R_r$, where the pairs $\langle k_r, R_r \rangle$ are precisely all possibilities from the following:

(i) $R_r = \text{PAR}(U_1{:}P_i,\ U_2{:}Q)$ and

$k_r = g_i = \text{SKIP}$

or $k_r = g_i = c!a$ and $c \in \text{oute}(U_1) - \text{ins}(U_2)$

or $k_r = g_i = c?x$ and $c \in \text{ins}(U_1) - \text{outs}(U_2)$

(ii) $R_r = \text{PAR}(U_1{:}P,\ U_2{:}Q_j)$ and

$k_r = h_j = \text{SKIP}$

or $k_r = h_j = c!e$ and $c \in \text{outs}(U_2) - \text{ins}(U_1)$

or $k_r = h_j = c?x$ and $c \in \text{ins}(U_2) - \text{outs}(U_1)$

(iii) $R_r = \text{SEQ}(x{:=}e,\ \text{PAR}(U_1{:}P_i,\ U_2{:}Q_j))$

$k_r = \text{SKIP}$ and

$g_i = c!e$ and $h_j = c?x$ and $c \in \text{ine}(U_2) \cap \text{outs}(U_1)$

or $g_i = c?x$ and $h_j = c!e$ and $c \in \text{ins}(U_1) \cap \text{outs}(U_2)$.

<expansion 2>

(i) and (ii) above represent P and Q (respactively) making independent progress. (iii) represents the effecte of communication between P and Q.

6. Laws of declaration

The construct VAR $x_1 \ldots x_n$:P declares the variables $x_1 \ldots x_n$ for use within P. These variables are distinct from any other variables with the same names that may be present in the external scope. It does not matter whether variables are declared in one list or singly:

(6.1) $$\text{VAR } x_1\text{:(VAR } x_2\text{: } \ldots \text{ VAR } x_n\text{:P)}\ldots\text{)} = \text{VAR } x_1 \ldots x_n\text{:P}$$ ⟨VAR assoc⟩

Nor does it matter in which order they are declared.

(6.2) $$\text{VAR } x_1\text{:(VAR } x_2\text{:P)} = \text{VAR } x_2\text{:(VAR } x_1\text{:P)}$$ ⟨VAR sym⟩

If a declared variable is never used, its declaration has no effect.

(6.3) $$\text{VAR x:P} = \text{P} \quad \text{if } x \notin \text{free(P)}$$ ⟨VAR elim⟩

One can change the name of a bound variable, provided the new name is not already used for a free variable.

(6.4) $$\text{VAR x:P} = \text{VAR y:P}\left[{}^{y}\!/\!_{x}\right] \quad \text{if } y \notin \text{free(P)}$$ ⟨VAR rename⟩

(Note that any clashes of y with bound variables of P are dealt with by the renaming implicit in the substitution operator.)

Generally speaking, the scope of a bound variable may be increased without effect, provided it does not interfere with another variable with the same name. Thus each of the occam constructors has a distribution law with declaration. The first two say that if each component process of an IF or ALT declares the variable x, and that variable does not clash with the booleans or guards, then the declaration may be moved outside the constructor.

(6.5) $\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ (\text{VAR}\ x{:}P_i) = \text{VAR}\ x{:}(\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i)$

provided x is free in no g_i <VAR – ALT distrib>

(6.6) $\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ (\text{VAR}\ x{:}P_i) = \text{VAR}\ x{:}(\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ P_i)$

provided x is free in no b_i <VAR – IF distrib>

Note that it is possible to deal with cases where x is only declared in a few of the P_i, but is not free in any other, by using 6.3.

Two laws are required for SEQ, one for each of its arguments.

(6.7) $\text{SEQ}(\text{VAR}\ x{:}P, Q) = \text{VAR}\ x{:}\text{SEQ}(P,Q)$ if $x \notin$ free (Q) <VAR – SEQ 1>

(6.8) $\text{SEQ}(P, \text{VAR}\ x{:}Q) = \text{VAR}\ x{:}\text{SEQ}(P,Q)$ if $x \notin$ free (P) <VAR – SEQ 2>

The law for PAR takes into account the fact that, when a declaration is moved outside the constructor, the process that uses it must now declare the fact that it might want to use the variable declared.

(6.9) $\text{PAR}(U_1{:}(\text{VAR}\ x{:}P), U_2{:}Q) = \text{VAR}\ x{:}\ \text{PAR}(U_1^{\ddagger}{:}P_1, U_2{:}P_2)$,

<u>provided</u> x is not free in $U_2{:}P_2$, where $U_1^{\ddagger}$ is U_1 modified to include a declaration of the variable x (in the notation of [R], it is the union of U_1 and USING(VAR x)).

<VAR – PAR>

When a variable is used for inputting, the effect is the same as that of inputting to a completely new variable, and then assigning to the original ones.

(6.10) $\mathrm{ALT}(c?x\ P, \underset{\sim}{G}) = \mathrm{VAR}\ y{:}\ \mathrm{ALT}(c?y\ \mathrm{SEQ}(x{:=}y, P), \underset{\sim}{G})$

provided $x \neq y$ and y is not free in P or $\underset{\sim}{G}$ <input renaming>

There is no point in assigning to a variable at tha very end of its scope, since the value given to it can have no effect.

(6.11)* $\mathrm{VAR}\ x{:}(\langle x\rangle + \underset{\sim}{y}) := (\langle e\rangle + \underset{\sim}{f}) = \mathrm{VAR}\ x{:}(\underset{\sim}{y} := \underset{\sim}{f})$ <assignment elim>

The final law of VAR is required to deal with uses of uninitialised variables in expressions. Upon declaration a variable may take any value, the choice being nondeterministic. Its value remains constant until it is assigned or input to. Thus the value of one uninitialised variable may be replaced by that of another, provided it has not yet been read and the value of the second variable is used nowhere else.

(6.12) $\mathrm{VAR}\ x{:}P = \mathrm{VAR}\ x{:}\mathrm{SEQ}(\mathrm{VAR}\ z{:}(x{:=}z), P)$ <initialisation>

It turns out that we only need one law to deal with channel declarations: an elimination rule analogous to (6.3).

(6.13) $\mathrm{CHAN}\ c_1 \ldots c_n{:}P = P$ if none of $c_1 \ldots c_n$ appears free in P. <CHAN elim>

The reason for this simplicity is that our normal form will eliminate all PAR constructs, and hence all internal use of channels.

7. Laws of $\perp$

Recall that $\perp$ is the divergent process WHILE true SKIP. In practice this process may be considered broken, for not only will it never interact with the outside world, but what is worse the environment can never detect this fact. (Seeing that the process is still performing internal actions, an observer can never discount the possibility that it might still do something.) A divergent process can also be regarded as having the most undefined behaviour possible, since it forever performs internal actions in an effort to decide what its behaviour will be, but never makes any progress.

With this philosophy in mind, we postulate that the divergent process is the worst possible. Now, in general, if P's behaviour is more predictable than that of Q, we must regard P as better (since whenever Q will guarantee the success of some experiment, so will P). We are thus forced to identify $\bot$ with all processes that <u>might</u> diverge (before doing anything else). It is quite reasonable to make this identification: in practice, a process which can either behave correctly or diverge will probably do the former while it is being tested, but will do the latter when it is being used in earnest. Putting it more simply, a racing program is always a programming error and may be considered broken. We therefore choose the simplest and most convenient laws, which state that almost any program made from a broken component is itself broken.

Our philosophy gives rise to a number of laws. First, a process that can automatically choose to diverge must be identified with $\bot$.

(7.1)† $\mathrm{ALT}(\mathrm{SKIP}\,\bot,\ \underline{G}) = \bot$ <ALT-SKIP zero>

It is clear that, if the first operand of a SEQ construct can diverge, so can the whole construct.

(7.2)* $\mathrm{SEQ}(\bot,\ P) = \bot$ <SEQ left zero>

If the first operand of a SEQ terminates before interacting with its environment, divergence in the second argument yields divergence in the whole construct.

(7.3)* $SEQ(\underset{\sim}{x}:=\underset{\sim}{e}, \perp) = \perp$ <SEQ right zero>

Divergence in one operand of a PAR may give rise to divergence in the complete construct, since an implementation may choose to run one argument until it can proceed no further before running another.

(7.4)* $PAR(u_1: \perp, u_2:P) = \perp$ <PAR zero>

2. A pre-normal form

The first section introduced almost all the laws one requires to characterise the semantics of occam. Unfortunately it is not satisfactory merely to state this; we must find some way of demonstrating it. This is especially true because we already have a denotational semantics; we would like the laws to yield the same equivalences. Even if we had no standard semantics to characterise, it would still be necessary to investigate the structure of the classes of inter-transformable programs, because it is only this that reveals the true power of a set of laws.

As explained in the introduction, our method of demonstrating the power of our laws will be the discovery of a normal form for finite programs. Every such program will have a normal form equivalent (through transformation), but two normal form programs will have the same value in the denotational semantics only if there are (at most) trivial syntactic differences between them.

A normal form must therefore exactly capture our ideas about denotational equivalence. This gives rise to a number of interrelated problems, all of which need to be solved before we have a normal form.

a) We need to characterise a process' behaviour as a communicating agent. In other words, we must identify a unique way of representing each possible pattern of communication a process might exhibit. For example, if U_1 and U_2 are suitable parallel declarations, the processes

ALT(c?x d?y, d?y c?x) and

PAR(U_1:c?x, U_2:d?y)

are equivalent, and therefore have the same normal form.

b) we need to characterise, relative to its communicating behaviour, the ways in which a process assigns to its variables. For example, the following pair of programs have the same effect on the final state and so have the same normal form:

x:=3 and VARy:SEQ(y:=3, z:=z, x:=y, y:=6)

There are important distinctions that need to be made between processes at the boundary between (a) and (b). Consider the two processes

PAR(U_1:c!1, U_2:ALT(d?x STOP, c?x d?x))

and d?x (U_1 and U_2 are suitably chosen).

Both processes have exactly the same communicating behaviour (they input along channel d), and when they terminate they have the same effect on their free variable x. However, the first process is strictly less deterministic than the second: it is not obliged to terminate successfully; when composed in sequence with another process the second process need not be started.

c) The use of bound variables needs to be regularised. In writing a program, one often has a lot of freedom in the use of bound variables: not only in where they are declared, but also in whether to declare a new variable or re-use an old one. For example, the following pair of equivalent programs must have the same normal form.

SEQ(c?x, c?x, d!x)

and VAR y,z:SEQ(c?y, c?z, x:=z, d!z).

An essential aid to the solution of (a) and (b) above is a calculus for deciding the equivalence of expressions. For example, $2 + 2 = 4 \equiv$ <u>true</u>, and (x mod 3) + (x + 1 mod 3) + (x + 2 mod 3) $\equiv$ 3. Often we need to decide such equivalences in the context of the booleans representing the facts already known about the variables involved. For example the programs

```
IF
  x mod 2 = 0
     c!(x/2)*2
  x mod 2 = 1
     c!((x+1)/2)*2-1
```

and c!x

are equivalent, because of the equivalences of "x" with "(x/2)*2" and "((x+1)/2)*2-1" in the respective (boolean) contexts.

Because this issue, though important, is not really relevant to the algebraic properties of occam, we will abstract away from it. Specifically, we will assume a knowledge of all true facts of the form

$b_1 \models b_2$ for boolean expressions b_1 and b_2

meaning "in all states where b_1 is satisfied, so is b_2". Thus our later completeness results are <u>relative</u> to this knowledge.

Our approach has the advantage of not tying us to a particular syntax and semantics for the space of expressions. We do, however, make frequent demands on the syntax and semantics of expressions representing booleans, the good behaviour of expressions under substitution for their variables, and the fact that all expressions in occam are evaluated without side-effects and without fear of non-termination (even 27/0!).

The discovery of a full normal form is rather difficult. We therefore introduce an intermediate form to act as a conceptual and technical bridge. This will essentially solve the problems described in (b) and (c) above, as well as simplifying the most difficult problem, which is the one described in (a). The intermediate form is called IF/ALT form, because it eliminates all uses of SEQ and PAR. It has a single parameter: a list of free variables.

We will say that a program is in $\underset{\sim}{x}$ – IF/ALT form if it has one of the following forms.

$\bot$ the wholly undefined, divergent process.

$\underset{\sim}{x} := \underset{\sim}{e}$ a multiple (simultaneous) assignment to each free variable of $\underset{\sim}{x}$ (the parameter of the form).

$\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ P_i$ where each P_i is $\underset{\sim}{x}$ – IF/ALT and the b_i partition true (i.e. $b_1 \vee \ldots \vee b_n \equiv$ true, and $b_i \wedge b_j \equiv$ false whenever $i \neq j$). No variable free in the whole program is in any bound(P_i).

VAR $x_1, \ldots, x_m : \overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i$ where each P_i is $\underset{\sim}{x}$ – IF/ALT, each g_i has one of the forms SKIP, c!e or c?x_j. $\{x_1, \ldots, x_m\}$ are the (all distinct) variables used in guards of the third type. They are disjoint from each bound(P_i) and from the components of $\underset{\sim}{x}$. x_j can appear free in $g_i\ P_i$ only if g_i has the form c?x_j. No variable in $\underset{\sim}{x}$ or free in the whole program may be in any bound(P_i).

VAR x:P   where $x \in$ free(P) but x is not a component of $\underset{\sim}{x}$. P is $\underset{\sim}{x}$ – IF/ALT.

Note that all assignments in IF/ALT programs are final (i.e. occur at the end of a program's run, just before it terminates) and made only to free variables. Also, because of the way a fresh bound variable is created for every input, no variable that contains a value relevant to the program is overwritten until this final assignment. It is the introduction of multiple assignments that allows us to reduce the assignments in every program to this form. Not only do they bring symmetry by removing the order of assignments, but by allowing such assignments as

$$\langle x,y\rangle := \langle y,x\rangle$$

they will allow us to eliminate all assignments to bound variables.

Bound variables are of two types. The ones that are boclared as inputting variables are used only for input and subsequent use in expressions. Variables declared in programs of the final type (VAR x:P) can never be given a "proper" value (since they are neither input to nor assigned to). They are thus, purely and simply, uninitialised variables, which contain a nondaterministically chosen constant value throughout the life of P. Thus, in practice, all programs of this form would be regarded as erroneous.

The following is the main theorem of this section.

<u>Theorem 1</u>. If $\underset{\sim}{x}$ contains all the free veriables that the finite program P ever inputs or assigns to, then there is an $\underset{\sim}{x}$ - IF/ALT program P' such that $free(P') \subseteq free(P) \cup \underset{\sim}{x}$ and $P = P'$ is provable from the laws presented in section 1.

The proof of this theorem is that every such program can be transformed to $\underset{\sim}{x}$ - IF/ALT using the said laws. A strategy for performing this transformation is set out below.

The first step is to transform all SEQ and PAR constructs to binary applications (<SEQ-SKIP unit>(4.1), <PAR-SKIP unit> (5.1), <SEQ assoc> (4.2), <PAR assoc>(5.2)). ALT constructs are then unnesteo (<ALT assoc>(2.1), <ALT sym>(2.2)) and the boolean components of guards removed (<ALT sym>(2.2), <boolean guard elim> (2.4)). IF constructs are then unnested <IF assoc>(1.1).

The rest of the strategy is recursive. We deal in turn with each form a program might take.

The atomic processes are all straightforward:

STOP = ALT( ) <ALT - STOP unit> (2.3)

SKIP = $\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{x}$ <SKIP> (3.1), <identity assignment> (3.3)

x:=e = $\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{x}\left[{}^{e}/_{x}\right]$ <assignment sym> (3.2), <identity assignment> (3.3)

c!e = ALT(c!e $\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{x}$) <output> (2.7), <SKIP> (3.1), <identity assignment> (3.3)

c?x = VARy:ALT(c?y $\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{x}\left[{}^{y}/_{x}\right]$), where y is not a component of $\underset{\leftrightarrow}{x}$

<input> (2.6), <input renaming> (6.10), <identity assignment> (3.3), <SKIP> (3.1), <assignment sym> (3.2), <combine assignments> (4.7).

(Recall that, in IF/ALT, no free variables may be used for inputting.)

If the program P has the form $\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ P_i$, we recursively transform each P_i to $\underset{\leftrightarrow}{x}$ - IF/ALT, making sure (via <VAR rename> (6.4)) that the bound variables of the resulting programs do not collide with free(P). It only remains to make sure that the b_i partition <u>true</u> (<IF - STOP unit> (1.6), <IF priority> (1.2)) and transform any STOP thus introduced to ALT ( ) (<ALT - STOP unit> (3.3)).

If the program P has the form $\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i$, we recursively transform each P_i to $\underset{\leftrightarrow}{x}$ - IF/ALT P_i' (making sure that bound $(P_i') \cap$ free(P) $= \emptyset$). One then applies <input renaming> (6.10) to each of the input g_i in turn (choosing a suitable variable), and <VAR assoc> (6.1) to collapse the VARx's thus created to a single declaretion. The resulting program looks like

$$\text{VAR}x_1,\ldots,x_m:\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i'\ P_i''$$

where, if g_i = SKIP or c!e, $g_i' = g_i$ and $P_i" = P'$ and if g_i = c?x then g_i' = c?x_j and $P_i"$ = SEQ($x:=x_j$, P_i') for some j. The only thing left to do is to transform all the $P_i"$ of the second type to $\underset{\sim}{x}$ - IF/ALT. This is done by first transforming $x:=x_j$ to $\underset{\sim}{x}:=\underset{\sim}{x}\left[{}^{x_j}/_{x}\right]$ and then applying the procedure set out under SEQ below.

If the program has the form SEQ(P,Q) we recursively transform P and Q to $\underset{\sim}{x}$ - IF/ALT programs P' and Q'. We then apply the following recursive procedure which, given P' and Q' in $\underset{\sim}{x}$ - IF/ALT, transforms SEQ(P',Q') to $\underset{\sim}{x}$ - IF/ALT. The first step is to ensure (using <VAR rename> (6.4) if necessary) that free(P') $\cap$ bound(Q') = $\emptyset$ and vice-versa.

If P' = $\perp$ then SEQ(P',Q') = $\perp$ <SEQ left zero> (7.2)

If P' = $\overset{n}{\underset{i=1}{\text{IF}}}\ b_i\ P_i$ then SEQ(P',Q') =

$\overset{n}{\underset{i=1}{\text{IF}}}\ b_i$ SEQ (P_i,Q') <SEQ - IF distrib> (4.3); each

SEQ(P_i,Q') is dealt with recursively.

If P' = VAR $x_1 \ldots x_m$: $\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i$, then because free(Q') $\cap$ bound(P') = $\emptyset$, the declaration can be moved outside the SEQ (<VAR assoc>(6.1), <VAR - SEQ1> (6.7)) so that the program looks like

VAR $x_1 \ldots x_m$:SEQ($\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i$, Q')

We then apply <SEQ - ALT distrib> (4.4) to obtain

VAR $x_1 \ldots x_m$: $\overset{n}{\underset{i=1}{\text{ALT}}}\ g_i$ SEQ(P_i, Q')

and finally deal with the SEQ(P_i,Q') recursively.

If P' = VAR x:P", then because x $\notin$ free(Q'), the declaration can be moved outside the SEQ <VAR - SEQ 1>(6.7); we then appeal to recursion. The program will then have the form VARy:R. If y is not free in R its declaration can be removed with <VAR elim>(6.3).

If $P' = \underset{\sim}{x}:=\underset{\sim}{e}$ we need to deal with each case of Q' separately. If $Q' = \bot$, then $SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q') = \bot$ ⟨SEQ right zero⟩(7.3).

If $Q' = \underset{\sim}{x}:=\underset{\sim}{f}$, then $SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q') = \underset{\sim}{x}:=\underset{\sim}{f}\left[{}^{e}/_{x}\right]$ ⟨combine assignments⟩(4.7).

If $Q' = VARy{:}Q''$, then because of $y \notin \text{free}\ (\underset{\sim}{x}:=\underset{\sim}{e})$ we have $SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q') = VARy{:}SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q'')$ and can then appeal to recursion. The program will then have the form VARy:R. If y is not free in R then apply ⟨VAR − elim⟩(6.3).

If $Q' = \overset{n}{\underset{i=1}{IF}}\ b_i\ Q_i$, then, by ⟨assignment − IF distrib⟩(4.5) we have

$$SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q') = \overset{n}{\underset{i=1}{IF}}\ b_i\left[{}^{e}/_{x}\right] SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q_i).$$

We then deal with the $SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q_i)$ recursively, noting that the $b_i\left[{}^{e}/_{x}\right]$ partition <u>true</u>, because the b_i do.

If $Q' = VAR\ x_1\ \ldots\ x_m{:}ALT\ g_i\ Q_i$, the first step (noting that $\{x_1\ \ldots\ x_m\} \cap \text{free}\ (\underset{\sim}{x}:=\underset{\sim}{e}) = \emptyset$) is to move the declaration outside the SEQ to obtain

$$VAR\ x_1\ \ldots\ x_m{:}SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ \overset{n}{\underset{i=1}{ALT}}\ g_i\ Q_i).$$

Because the input variables of the g_i are the x_j, none of which appear in $\underset{\sim}{x}:=\underset{\sim}{e}$, we can use ⟨assignment − ALT distrib⟩(4.6) to get

$$VAR\ x_1\ \ldots\ x_m{:}\overset{n}{\underset{i=1}{ALT}}\ g_i\ SEQ(\underset{\sim}{x}:=\underset{\sim}{e},\ Q_i)$$

and then appeal to recursion.

Note that this procedure for reducing SEQ(P,Q), with P,Q already in $\underset{\sim}{x}$ − IF/ALT, is guaranteed to terminate because every recursive call strictly simplifies one of the two arguments, leaving the other one unchanged.

If we wish to transform VARy:P to $\underset{\sim}{x}$ - IF/ALT, the first step is to use <VAR rename> (6.4) if necessary to ensure that y is not a component of x. We then recursively transform P to an $\underset{\sim}{x}$ + <y> - IF/ALT program P'. Choosing a variable z that is distinct from y and does not appear in P, we use <initialisation> (6.12), <VAR - SEQ 1> (6.7), <VAR sym> (6.2) and <identity assignment> (3.3) to obtain

VARz:(VARy:SEQ($\underset{\sim}{x}$+<y>:=$\underset{\sim}{x}$+<z>), P')

We then apply the procedure for reducing sequential compositions of IF/ALT programs to reduce this to

VARz:(VARy:P")   where P" is $\underset{\sim}{x}$+<y> IF/ALT.

Observe that the only places y can appear in P" are on the left hand sides of the final multiple assignments, because the transformation from SEQ(($\underset{\sim}{x}$+<y>:=$\underset{\sim}{x}$+<z>),P') to P" replaces all others by z. (This is easy to prove by structural induction on P'.) We can therefore make repeated use of <VAR - ALT distrib> (6.5), <VAR - IF distrib> (6.6), <VAR sym> (6.2), <VAR assoc> (6.1) to shift the declaration VARy down to the leaves of P". It can be eliminated from those of the form VARy: $\perp$ by <VAR elim> (6.3), and leaves of the form VARy:$\underset{\sim}{x}$+<y> :=$\underset{\sim}{e}$+<f> are transformed to $\underset{\sim}{x}$:=$\underset{\sim}{e}$ by <assignment elim> (6.11) and <VAR elim> (6.3).

The resulting program is then just VARz:P^*, where P^* is the program obtained from P" by deleting all assignments to y. If z is not free in P^* we make use of <VAR elim> (6.3). In any case we are left with our desired $\underset{\sim}{x}$ - IF/ALT program, in which we note that y is not free.

If a program has the form CHAN $c_1 \ldots c_n$:P, we first recursively transform P to an $\underset{\sim}{x}$ - IF/ALT program P'. Now any occurrences of $c_1, \ldots, c_n$ within CHAN $c_1 \ldots c_n$:P' (other than their declaration) are syntactically incorrect - for P' contains no PAR constructs and so there is no place for internal communications on these channels. Since we have postulated that all programs are syntactically correct, we can infer that none of $c_1,\ldots,c_n$ appears free in P'. Thus ⟨CHAN elim⟩ (6.13) is applicable.

The only case that remains is that of PAR. It is important to note that none of the clauses we have so far dealt with have introduced a PAR construct (SEQ, on the other hand, was introduced by ALT and VAR). Thus the procedure we have already set up will work when given a program not containing any PAR constructs.

If we are given a program of the form PAR(U_1:P, U_2:Q), the first step is to recursively transform P into $\underset{\sim}{x}_1$-IF/ALT P' and Q into $\underset{\sim}{x}_2$-IF/ALT Q' where $\underset{\sim}{x}_1$ and $\underset{\sim}{x}_2$ are respectively the components of $\underset{\sim}{x}$ declared in U_1 and U_2. (That this transformation is possible follows from the correctness of PAR(U_1:P, U_2:Q).) PAR(U_1:P', U_2:Q') is then transformed to $\underset{\sim}{x}$-IF/ALT using the recursive procedure set out below. The first step is to make sure the bound variable sets of P' and Q' are disjoint from free(PAR(U_1:P', U_2:Q')) and the components of $\underset{\sim}{x}$. If either P' or Q' is $\perp$, we can apply ⟨PAR zero⟩(7.4) (and perhaps ⟨PAR sym⟩(5.3)) to obtain $\perp$.

If P' is $\underset{i=1}{\overset{n}{\text{IF}}}\ b_i\ P_i$, then since the b_i partition <u>true</u> we can apply ⟨PAR - IF distrib⟩ (5.4) to obtain

$$\underset{i=1}{\overset{n}{\text{IF}}}\ b_i\ \text{PAR}\ (U_1{:}P_i,\ U_2{:}Q').$$

We then recursively reduce each PAR(U_1:P_i, U_2:Q').

If Q' is $\underset{i=1}{\overset{n}{\text{IF}}}\ b_i\ Q_i$, then we apply ⟨PAR sym⟩(5.3) and then the above.

If P' is VARy:P" then since, by construction, y is not free in U_2:Q',
we can use <VAR − PAR>(6.9) to obtain

VARy:PAR(U_1^*:P", U_2:Q')

where U_1^* is U_1 with y "added"; we then appeal to recursion. If Q' is VARy:Q" we apply <PAR sym>(5.3) and the above. As before, if y is not free in the resulting body, its declaration can be removed by <VAR elim>(6.3).

If P' is $\underset{\sim}{x}_1 := \underset{\sim}{e}_1$ and Q' is $\underset{\sim}{x}_2 := \underset{\sim}{e}_2$ then, noting that the elements of $\underset{\sim}{x}_1$ and $\underset{\sim}{x}_2$ are disjoint subsets of those of $\underset{\sim}{x}$, we can apply <PAR assignments>(5.5), <identity assignment> (3.3) and <assignment sym>(3.2) to obtain something of the form $\underset{\sim}{x} := \underset{\sim}{e}$.

If P' is VAR $y_1 \ldots y_m$:$\underset{i=1}{\overset{n}{\text{ALT}}}$ g_i P_i and Q' is $\underset{\sim}{x}_2 = \underset{\sim}{e}_2$, then by construction none of $y_1 \ldots y_m$ appear free in U_2:Q', so the VAR may be moved outside the PAR, using <VAR assoc>(6.1) and <VAR − PAR>(6.9) (thereby changing U_1 to U_1^*, say). We can then use <expansion 1>(5.6) to transform it to something of the form

VAR $y_1 \ldots y_m$: $\underset{i \in x}{\text{ALT}}$ g_i PAR(U_1^*: P_i, U_2:Q').

The y_i that no longer appear as input variables among the g_j still appear in the declaration and in U_1^*. They are removed by first moving them inside the ALT (<VAR assoc>(6.1), <VAR sym>(6.2), <VAR − ALT distrib>(6.5) and then inside the PARs <VAR − PAR>(6.9), removing them from U_1^* (obtaining U_1', say). Because these variables are free in no remaining P_i, we can finally delete their declarations using <VAR elim>(6.3). When we have recursively transformed the resulting PAR(U_1':P_i, U_2:Q'), the whole program is $\underset{\sim}{x}$ − IF/ALT.

The symmetric case ($P' = \underset{\sim}{x}_1 := \underset{\sim}{e}_1$, $Q' = \text{VAR } y_1 \ldots y_m : \overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ Q_i$) is dealt with by the above, after applying <PAR sym>(5.3).

The only remaining case is when $P' = \text{VAR } y_1 \ldots y_m : \overset{n}{\underset{i=1}{\text{ALT}}}\ g_i\ P_i$ and $Q' = \text{VAR } z_1 \ldots z_s : \overset{t}{\underset{i=1}{\text{ALT}}}\ h_i\ Q_i$. The same type of strategy as above, using <expansion 2>(5.7), will transform PAR(U_1:P', U_2:Q') to something of the form

$$\text{VAR } x_1' \ldots x_p' : \overset{N}{\underset{i=1}{\text{ALT}}}\ k_i\ R_i$$

where there is some M ($0 \leq M \leq N$) such that $1 \leq i \leq M$ implies k_i is SKIP and R_i is VAR y_i':SEQ($y_i' := e_i$, R_i') where R_i' is $\underset{\sim}{x}$ - IF/ALT; $M < i \leq N$ implies R_i is $\underset{\sim}{x}$-IF/ALT. It can further be guaranteed that the x_i' are precisely the (distinct) variables used for input among the k_i ($i > M$), and that no x_i' or y_i' occurs in any R_j except the one obviously corresponding to it. (The first M guarded processes result from communications between P' and Q', the rest from independent action by either P' or Q'.)

Observing that no R_i ($1 \leq i \leq M$) has any occurrence of PAR, we can safely transform them to $\underset{\sim}{x}$ - IF/ALT. This having been done the whole program is in $\underset{\sim}{x}$- IF/ALT, as required, after perhaps some renaming of bound variables. (Care is required over this last point because we have no reason for supposing that the programs R_i are in any sense "simpler" than the complete program. It is therefore vital that this transformation does not introduce a PAR and so make use of the recursive procedure we are currently defining.)

This completes the description of the procedure for transforming PAR(U_1:P, U_2:Q) to $\underset{\sim}{x}$ - IF/ALT. Since that was the last clause of the main procedure, we have also completed the description of how to transform a general program to IF/ALT.

Syntactic approximation

Finite programs are relatively easy to reason about algebraically, but do not tend to be very useful in practice. Fortunately there are techniques which allow us to apply our results on finite programs to general programs: syntactic approximation allows us to identify every program with a set of finite ones.

The concept of syntactic approximation is quite well known (see, for example, [G]) and has been applied to CSP in similar circumstances to ours [B]. It gives a pre-order (in our case a partial order) on the syntax of a language. The order is a very simple one, based on the ideas that replacing part of a program by the least defined program (in our case $\perp$) produces an approximation, and that unfolding a recursion (in our case a WHILE loop) produces an approximation.

Through most of this paper we make no formal distinction between the text of a program and its value (semantics). However when considering syntactic approximation it is necessary to make a clear distinction: we will therefore place quotes ($\ulcorner P \urcorner$) round any program that is to be considered as a syntactic object, and continue to use unadorned programs (P) for the corresponding semantic values. It is important to note that $P = Q$ does not imply $\ulcorner P \urcorner = \ulcorner Q \urcorner$, so the clauses below may not be combined with our existing laws (which are all semantic).

We will write $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner$ if $\ulcorner P \urcorner$ is a syntactic approximation to $\ulcorner Q \urcorner$. The following clauses define $\leqslant$ for our version of occam.

1) $\ulcorner \bot \urcorner \leqslant \ulcorner P \urcorner$

2) $\ulcorner P \urcorner \leqslant \ulcorner P \urcorner$

3) $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner \;\; \& \;\; \ulcorner Q \urcorner \leqslant \ulcorner R \urcorner \Longrightarrow \ulcorner P \urcorner \leqslant \ulcorner R \urcorner$

4) $\bigwedge_{i=1}^{n} \ulcorner P_i \urcorner \leqslant \ulcorner Q_i \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{SEQ}}}\ P_i \urcorner \leqslant \ulcorner \underset{i=1}{\overset{n}{\mathrm{SEQ}}}\ Q_i \urcorner$

5) $\bigwedge_{i=1}^{n} \ulcorner P_i \urcorner \leqslant \ulcorner Q_i \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{PAR}}}\ U_i{:}P_i \urcorner \leqslant \ulcorner \underset{i=1}{\overset{n}{\mathrm{PAR}}}\ U_i{:}Q_i \urcorner$

6)* $\bigwedge_{i=1}^{n} \ulcorner C_i \urcorner \leqslant^{c} \ulcorner C_i' \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{IF}}}\ C_i \urcorner \leqslant \ulcorner \underset{i=1}{\overset{n}{\mathrm{IF}}}\ C_i' \urcorner$

7)* $\bigwedge_{i=1}^{n} \ulcorner G_i \urcorner \leqslant^{g} \ulcorner G_i' \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{ALT}}}\ G_i \urcorner \leqslant \ulcorner \underset{i=1}{\overset{n}{\mathrm{ALT}}}\ G_i' \urcorner$

8) $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner \Longrightarrow \ulcorner \mathrm{VAR}\ x_1 \ldots x_n{:}P \urcorner \leqslant \ulcorner \mathrm{VAR}\ x_1 \ldots x_n{:}Q \urcorner$

9) $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner \Longrightarrow \ulcorner \mathrm{CHAN}\ c_1 \ldots c_n{:}P \urcorner \leqslant \ulcorner \mathrm{CHAN}\ c_1 \ldots c_n{:}Q \urcorner$

10) $\ulcorner \mathrm{IF}(b\ \mathrm{SEQ}(P, \mathrm{WHILE}\ b\ P), \neg b\ \mathrm{SKIP}) \urcorner \leqslant \ulcorner \mathrm{WHILE}\ b\ P \urcorner$

* Clauses (6) and (7) require the definition of auxiliary relations $\leqslant^{c}$ and $\leqslant^{g}$ on (respectively) conditionals and guarded processes. These satisfy

11) $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner \Longrightarrow \ulcorner b\ P \urcorner \leqslant^{c} \ulcorner b\ Q \urcorner$

12) $\bigwedge_{i=1}^{n} \ulcorner C_i \urcorner \leqslant^{c} \ulcorner C_i' \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{IF}}}\ C_i \urcorner \leqslant^{c} \ulcorner \underset{i=1}{\overset{n}{\mathrm{IF}}}\ C_i' \urcorner$

13) $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner \Longrightarrow \ulcorner g\ P \urcorner \leqslant^{g} \ulcorner g\ Q \urcorner$

14) $\bigwedge_{i=1}^{n} \ulcorner G_i \urcorner \leqslant^{g} \ulcorner G_i' \urcorner \Longrightarrow \ulcorner \underset{i=1}{\overset{n}{\mathrm{ALT}}}\ G_i \urcorner \leqslant^{g} \ulcorner \underset{i=1}{\overset{n}{\mathrm{ALT}}}\ G_i' \urcorner .$

Formally, $(\leqslant, \leqslant^{c}, \leqslant^{g})$ is the smallest triple of relations satisfying (1–14). $\leqslant$ is a partial order on the syntax of our language. (This can fail for other languages if they have more general forms of recursion: one can have distinct pieces of syntax $\ulcorner P \urcorner$ and $\ulcorner Q \urcorner$ such that $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner$ and $\ulcorner Q \urcorner \leqslant \ulcorner P \urcorner$, e.g. $\mu p.\mu q.p$ and $\mu q.\mu p.\mu q.p$.) It is important to remember that $\leqslant$ is a purely syntactic relation, and that it is <u>not</u> permissible to use the above clauses in conjunction with our laws (which preserve semantics rather than syntax).

FIN($\ulcorner P \urcorner$), the set of P's finite syntactic approximations, is defined to be $\left\{\ulcorner Q \urcorner \,\middle|\, \ulcorner Q \urcorner \leqslant \ulcorner P \urcorner \text{ and } \ulcorner Q \urcorner \text{ is finite}\right\}$. It is easy to write down an equivalent definition of FIN($\ulcorner P \urcorner$) that is a straightforward recursion on syntax. Typical clauses are given below (the only moderately difficult one being WHILE).

$$\mathrm{FIN}(\ulcorner \underset{\sim}{x} := \underset{\sim}{e} \urcorner) = \left\{\ulcorner \bot \urcorner, \ulcorner \underset{\sim}{x} := \underset{\sim}{e} \urcorner\right\}$$

$$\mathrm{FIN}(\ulcorner c?x \urcorner) = \left\{\ulcorner \bot \urcorner, \ulcorner c?x \urcorner\right\}$$

$$\mathrm{FIN}(\ulcorner \overset{n}{\underset{i=1}{\mathrm{SEQ}}}\, P_i \urcorner) = \left\{\ulcorner \bot \urcorner\right\} \cup \left\{\ulcorner \overset{n}{\underset{i=1}{\mathrm{SEQ}}}\, Q_i \urcorner \,\middle|\, \bigwedge_{i=1}^{n} \ulcorner Q_i \urcorner \in \mathrm{FIN}(\ulcorner P_i \urcorner)\right\}$$

$$\begin{aligned}\mathrm{FIN}(\ulcorner \mathrm{WHILE}\ b\ P \urcorner) = {} & \left\{\ulcorner \bot \urcorner, \ulcorner \mathrm{IF}(b\ \bot,\ \neg b\ \bot) \urcorner, \ulcorner \mathrm{IF}(b\ \bot,\ \neg b\ \mathrm{SKIP}) \urcorner\right\} \\ & \cup \left\{\ulcorner \mathrm{IF}(b\ \mathrm{SEQ}(Q_1, Q_2),\ \neg b\ \bot) \urcorner,\right. \\ & \quad \ulcorner \mathrm{IF}(b\ \mathrm{SEQ}(Q_1, Q_2),\ \neg b\ \mathrm{SKIP}) \urcorner \,\Big|\, \\ & \quad\quad \left.\ulcorner Q_1 \urcorner \in \mathrm{FIN}(\ulcorner P \urcorner), \ulcorner Q_2 \urcorner \in \mathrm{FIN}(\ulcorner \mathrm{WHILE}\ b\ P \urcorner)\right\}\end{aligned}$$

(The last clause, which is circular, is easily seen to have a unique solution.)

Any finite, non divergent, behaviour of a program has required only finitely many iterations of any loop. It is therefore possible to unwind the program that many times, obtaining a finite syntactic approximation which exhibits the same behaviour. Of course, any non divergent behaviour possible for a syntactic approximation will also be possible for the original process. Intuitively, there is thus a close relationship between the behaviour of a process and those of its finite syntactic approximations. To understand this relationship properly we need to go back to our underlying semantic model.

The denotational semantics of $[R]$ map each process into a domain with a partial order according to which one process is greater than another if it is better defined, or more predictable. If P and Q are processes, we will write $P \sqsubseteq Q$ (Q is more deterministic than P) if the semantic value of P is less than that of Q for all environments with unbounded sets of free locations and channels, and states where unused locations are mapped to <u>error</u>. $P \sqsubseteq Q$ is equivalent to

$$P \;=\; \text{ALT}(\text{SKIP } P,\ \text{SKIP } Q).$$

This law simply says that every behaviour of Q is also possible for P; thus in observing Q we cannot be sure that we are not looking at P. $\sqsubseteq$ induces a natural partial order on occam terms (factored under the equivalence induced by the domain).

The following three lemmas express the formal properties we will require of syntactic approximations. The first one is easy to prove (in the denotational semantics) by structural induction.

<u>Lemma 1</u> If $\ulcorner P \urcorner \leqslant \ulcorner Q \urcorner$, then $P \sqsubseteq Q$.

Of course, the converse to Lemma 1 does not hold.

The second lemma is easy to prove using a combination of structural induction and mathematical induction (the latter for WHILE loops).

Lemma 2 $FIN(\ulcorner P \urcorner)$ is (under $\leq$) a directed set (i.e. if $\ulcorner Q_1 \urcorner, \ulcorner Q_2 \urcorner \in FIN(\ulcorner P \urcorner)$, there is some $\ulcorner Q \urcorner \in FIN(\ulcorner P \urcorner)$ with $\ulcorner Q_1 \urcorner \leq \ulcorner Q \urcorner$ and $\ulcorner Q_2 \urcorner \leq \ulcorner Q \urcorner$).

Lemmas 1 and 2 tell us that the semantic values of the elements of $FIN(\ulcorner P \urcorner)$ are themselves a directed set under $\sqsubseteq$. The last, and most important, of our lemmas, shows just how this set characterises the samantics of P. It, also, is proved using a combination of structural and mathematical induction.

Lemma 3 $\{Q \mid \ulcorner Q \urcorner \in FIN(\ulcorner P \urcorner)\}$ is a directed set (under $\sqsubseteq$) with least upper bound P (i.e. $\bigsqcup \{Q \mid \ulcorner Q \urcorner \in FIN(\ulcorner P \urcorner)\} = P$).

Later we will take advantage of this strong way in which the semantic value of a process is determined by its syntactic approximations.

Proving additional laws

One very useful consequence of Lemma 3 above is that, if we want to prove a new algebraic law, it will usually be sufficient to prove it for finite programs. For example, consider the law

$$SEQ(P,\ SEQ(Q,R)) \ = \ SEQ(SEQ(P,Q),R).$$

This (the conventional binary associative law of SEQ) is not trivially deducible from our existing laws, even though it is semantically true. However, suppose we have proved it for all finite P, Q, R. (We will shortly do this.) Then, using Lemma 3, we have for general P, Q, R:

$$SEQ(P,SEQ(Q,R)) = \bigsqcup \{F \mid \ulcorner F \urcorner \in FIN(\ulcorner SEQ(P,SEQ(Q)R)) \urcorner\}$$

Now because the few elements F of the first set which are not of the form SEQ(P',SEQ(Q',R')) are easily proved (using the laws) equivalent to ones that are, using the laws, e.g.

$$SEQ(P,\bot) = SEQ(P, SEQ(\bot,\bot)) \text{ this is equal to}$$

$$\bigsqcup\{SEQ(P', SEQ(Q',R')) \mid \ulcorner P'\urcorner \in FIN(\ulcorner P\urcorner), \ulcorner Q'\urcorner \in FIN(\ulcorner Q\urcorner), \ulcorner R'\urcorner \in FIN(\ulcorner R\urcorner)\}.$$

By our assumption that the result holds for finite processes this in turn is equal to

$$\bigsqcup\{SEQ(SEQ(P',Q'),R') \mid \ulcorner P'\urcorner \in FIN(\ulcorner P\urcorner), \ulcorner Q'\urcorner \in FIN(\ulcorner Q\urcorner), \ulcorner R'\urcorner \in FIN(\ulcorner R\urcorner)\}$$
$$= \bigsqcup\{F \mid \ulcorner F\urcorner \in FIN(\ulcorner SEQ(SEQ(P,Q),R)\urcorner)\}$$
$$= SEQ(SEQ(P,Q),R).$$

Since we are in the process of setting up powerful machinery for dealing with finite programs (for example Theorem 1) there are advantages in only having to prove new laws for them. In particular, it is enough to prove them for IF/ALT programs (since, by Theorem 1, every finite program is equivalent to one in IF/ALT). As an illustration of the techniques one can employ to prove laws for IF/ALT programs, we will complete the proof of the SEQ associativity law given above. By virtue of what we have already established, the following proposition will suffice.

<u>Proposition</u> If P,Q,R are all $\underset{\sim}{x}$-IF/ALT, then

$$SEQ(P,SEQ(Q,R)) = SEQ(SEQ(P,Q),R).$$

<u>Proof</u> We use structural induction on the triple (P,Q,R). Suppose the result holds for all simpler triples (P',Q',R'). ((P',Q',R') is <u>simpler</u> than (P,Q,R) if each of its components is a (not necessarily proper) syntactic subcomponent of the corresponding component of (P,Q,R), except possibly for changes of variables not in $\underset{\sim}{x}$. At least one must be a proper subcomponent.)

If $P = \bot$ the result is trivial by applications of <SEQ left zero>(7.2).

If $P = \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ P_i$, we have

$$\begin{aligned}
\mathrm{SEQ}(P,\mathrm{SEQ}(Q,R)) &= \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ \mathrm{SEQ}(P_i,\ \mathrm{SEQ}(Q,R)) && \langle \mathrm{SEQ-IF\ distrib}\rangle (4.3)\\
&= \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ \mathrm{SEQ}(\mathrm{SEQ}(P_i,Q),R) && \text{(by induction)}\\
&= \mathrm{SEQ}(\overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ \mathrm{SEQ}(P_i,Q),R) && \langle \mathrm{SEQ-IF\ distrib}\rangle (4.3)\\
&= \mathrm{SEQ}(\mathrm{SEQ}(\overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ P_i,\ Q),\ R) && \langle \mathrm{SEQ-IF\ distrib}\rangle (4.3)\\
&= \mathrm{SEQ}(\mathrm{SEQ}(P,Q),\ R) && \text{as required.}
\end{aligned}$$

If $P = \mathrm{VAR}x{:}P'$ we first ensure (via <VAR rename>(6.4)) that x is not in $\mathrm{free}(Q) \cup \mathrm{free}(R)$, and then

$$\begin{aligned}
\mathrm{SEQ}(P,\mathrm{SEQ}(Q,R)) &= \mathrm{VAR}x{:}\mathrm{SEQ}(P',\mathrm{SEQ}(Q,R)) && \langle \mathrm{VAR-SEQ\ 1}\rangle\ (6.7)\\
&= \mathrm{VAR}x{:}\mathrm{SEQ}(\mathrm{SEQ}(P',Q),R) && \text{(induction)}\\
&= \mathrm{SEQ}(\mathrm{SEQ}(P,Q),R) && \langle \mathrm{VAR-SEQ\ 1}\rangle\ (6.7)\ \text{twice.}
\end{aligned}$$

If $P = \mathrm{VAR}x_1 \ldots x_m{:}\overset{n}{\underset{i=1}{\mathrm{ALT}}}\ g_i\ P_i$ one combines the techniques of the previous two cases (ueing <SEQ - ALT distrib>(4.4) rather than <SEQ - IF distrib>(4.3)).

If $P = \underline{x}:=\underline{e}$ we need to deal with the individual cases of Q separately. If $Q = \bot$ the result is trivial by <SEQ left zero>(7.2) and <SEQ right zero>(7.3).

If $Q = \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ Q_i$ then

$$\begin{aligned}
\mathrm{SEQ}(P,\mathrm{SEQ}(Q,R)) &= \mathrm{SEQ}(\underline{x}:=\underline{e},\ \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\ \mathrm{SEQ}(Q_i,\ R))\\
&\qquad \langle \mathrm{SEQ-IF\ distrib}\rangle (4.3)\\
&= \overset{n}{\underset{i=1}{\mathrm{IF}}}\ b_i\left[\underline{e}/\underline{x}\right] \mathrm{SEQ}(\underline{x}:=\underline{e},\ \mathrm{SEQ}(Q_i,R))\\
&\qquad \langle \mathrm{assignment-IF\ distrib}\rangle (4.5)
\end{aligned}$$

$$= \mathop{IF}_{i=1}^{n} b_i \left[{}^{\underset{\leftrightarrow}{e}}/_{\underset{\leftrightarrow}{x}}\right] \mathrm{SEQ}(\mathrm{SEQ}(\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{e},\ Q_i),\ R)$$

(induction)

$$= \mathrm{SEQ}(\mathop{IF}_{i=1}^{n} b_i \left[{}^{\underset{\leftrightarrow}{e}}/_{\underset{\leftrightarrow}{x}}\right] \mathrm{SEQ}(\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{e},\ Q_i),\ R)$$

<SEQ - IF distrib> (4.3)

$$= \mathrm{SEQ}(\mathrm{SEQ}(\underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{e},\ \mathop{IF}_{i=1}^{n} b_i\ Q_i),\ R)$$

<assignment - IF distrib> (4.5)

$$= \mathrm{SEQ}(\mathrm{SEQ}(P,Q),\ R)\ .$$

If $Q = \mathrm{VAR}x{:}Q'$ the result may be established (after possible renaming of bound variables) by <VAR - SEQ 1,2> (6.7, 6.8) and induction.

If $Q = \mathrm{VAR}x_1\ldots x_m{:}\mathop{ALT}_{i=1}^{n} g_i\ Q_i'$ the result follows using the techniques of the previous two clauses, using <SEQ - ALT distrib> (4.4) in place of <SEQ - IF distrib> (4.3) and <assignment - ALT distrib> (4.6) in place of <assignment - IF distrib> (4.5).

If $Q = \underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{f}$ we need to consider each case of R separately. If $R = \perp$ the result follows simply from <SEQ right zero> (7.3) and <combine assignments> (4.7). If $R = \underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{f}'$ we have

$$\mathrm{SEQ}(P,\mathrm{SEQ}(Q,R)) = \underset{\leftrightarrow}{x}:=(\underset{\leftrightarrow}{f}'\left[{}^{\underset{\leftrightarrow}{f}}/_{\underset{\leftrightarrow}{x}}\right])\left[{}^{\underset{\leftrightarrow}{e}}/_{\underset{\leftrightarrow}{x}}\right] \quad \text{<combine assignments> (4.7)}$$

$$= \underset{\leftrightarrow}{x}:=\underset{\leftrightarrow}{f}'\left[{}^{\underset{\leftrightarrow}{f}\left[{}^{\underset{\leftrightarrow}{e}}/_{\underset{\leftrightarrow}{x}}\right]}/_{\underset{\leftrightarrow}{x}}\right]$$

by properties of substitution

$$= \mathrm{SEQ}(\mathrm{SEQ}(P,Q),\ R) \quad \text{<combine assignments> (4.7).}$$

If $R = \mathrm{VAR}x_1\ldots x_m{:}\mathrm{ALT}\ g_i\ R_i$, then after possibly renaming $x_1\ldots x_m$ to avoid clashes with free(P) $\cup$ free(Q) we have

$$\text{SEQ}(P,\text{SEQ}(Q,R)) = \text{VAR}\,x_1\ldots x_m:\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ \text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{f},\ \underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\ R_i))$$

⟨VAR expansion⟩(6.1), ⟨VAR − SEQ 2⟩(6.8)

$$= \text{VAR}\,x_1\ldots x_m:\underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\left[\underset{\sim}{f}/\underset{\sim}{x}\right]\left[\underset{\sim}{e}/\underset{\sim}{x}\right]\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ \text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{f},\ R_i))$$

⟨assignment − ALT distrib⟩ (4.6) twice

$$= \text{VAR}\,x_1\ldots x_m:\underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\left[\underset{\sim}{f}\left[\underset{\sim}{e}/\underset{\sim}{x}\right]\Big/\underset{\sim}{x}\right]\text{SEQ}(\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ \underset{\sim}{x}:=\underset{\sim}{f}),\ R_i)$$

(induction and properties of substitution)

$$= \text{VAR}\,x_1\ldots x_m:\underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\left[\underset{\sim}{f}\left[\underset{\sim}{e}/\underset{\sim}{x}\right]\Big/\underset{\sim}{x}\right]\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{f}\left[\underset{\sim}{e}/\underset{\sim}{x}\right],\ R_i)$$

⟨combine assignments⟩ (4.7)

$$= \text{VAR}\,x_1\ldots x_m:\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{f}\left[\underset{\sim}{e}/\underset{\sim}{x}\right],\ \underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\ R_i)$$

⟨assignment − ALT distrib⟩ (4.6)

$$= \text{VAR}\,x_1\ldots x_m:\text{SEQ}(\text{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ \underset{\sim}{x}:=\underset{\sim}{f}),\ R)$$

⟨combine assignments⟩(4.7)

$$= \text{SEQ}(\text{SEQ}(P,Q),\ R)$$

⟨VAR expansion⟩(6.1), ⟨VAR − SEQ 2⟩(6.8)

If $R = \text{IF}\ b_i\ R_i$ the same argument as above applies, only ⟨assignment − IF distrib⟩(4.5) is used in place of ⟨assignment − ALT distrib⟩(4.6). The case of R = VARx:R' is easy.

This completes the proof.

Other laws can be proved in much the same way (often rather more easily). Some examples are given below.

a) $SEQ(SKIP, P) = SEQ(P, SKIP) = P$

b) $SEQ(P, \overset{n}{\underset{i=1}{IF}}\ b_i\ Q_i) = \overset{n}{\underset{i=1}{IF}}\ b_i\ SEQ(P, Q_i)$

if $b_1 \vee \ldots \vee b_n \equiv$ <u>true</u> and no variable in any b_i is altered by P.

c) $PAR(U_1{:}P,\ U_2{:}SKIP) = PAR(U_1{:}P) = P$

provided U_1 declares all global variables and channels used by P, and U_2 declares none of them.

Not all proofs of new laws go along these lines. Some may require the full power of a normal form, while some can be derived directly. As an example of direct derivation we here prove a law relating IF and ALT that is apparently more powerful than the law <IF – ALT distrib> (2.10) we already have.

$$\overset{n}{\underset{i=1}{ALT}}\ g_i\ (\overset{m}{\underset{j=1}{IF}}\ b_j\ P_{ij}) = \overset{m}{\underset{j=1}{IF}}\ b_j\ (\overset{n}{\underset{i=1}{ALT}}\ g_i\ P_{ij})$$

providing $b_1 \vee \ldots \vee b_m \equiv$ <u>true</u> and no variable input in a g_i appears in a b_j. <ALT – IF distrib>

This says that, providing the execution of the guards g_i always leads to the evaluation of the same conditionals, the value of which is not affected by the g_i, then the conditional choice may be brought outside.

To derive this law we first establish the following law as a lemma:

$$\overset{n}{\underset{i=1}{IF}}\ b_i\ P_i = \overset{n}{\underset{i=1}{IF}}\ b_i^*\ (IF\ b_i^*\ P_i)$$

where $b_i^* = \neg b_1 \wedge \ldots \wedge \neg b_{i-1} \wedge b_i$.

The right hand side may be transformed to $\mathop{\mathrm{IF}}\limits_{i=1}^{n} b_i^* \, p_i$ by repeated use of <$\wedge$- IF distrib> (1.8), <IF assoc>(1.1) and <IF sym>(1.3). It is then equivalent to the left hand side by <IF priority>(1.2).

The proof of <ALT - IF distrib> is as follows.

$$\mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \mathop{\mathrm{IF}}\limits_{j=1}^{m} b_j \, p_{ij}$$

$$= \mathrm{IF}\ b_1 \vee \ldots \vee b_m \left(\mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \mathop{\mathrm{IF}}\limits_{j=1}^{m} b_j \, p_{ij}\right)$$

(by <IF - <u>true</u> unit> (1.7), as $b_1 \vee \ldots \vee b_m \equiv \underline{true}$)

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k \left(\mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \mathop{\mathrm{IF}}\limits_{j=1}^{m} b_j \, p_{ij}\right)$$

(by <IF - $\vee$ distrib>(1.4))

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k^* \left(\mathrm{IF}\ b_k^* \left(\mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \mathop{\mathrm{IF}}\limits_{j=1}^{m} b_j^* \, p_{ij}\right)\right)$$

where $b_i^* = \neg b_1 \wedge \ldots \wedge \neg b_{i-1} \wedge b_i$ (by lemma)

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k^* \left(\mathrm{IF}\ b_k^* \mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \left(\mathrm{IF}\ b_k^* \left(\mathop{\mathrm{IF}}\limits_{j=1}^{m} b_j^* \, p_{ij}\right)\right)\right)$$

(by <IF - ALT distrib>(2.10), since no variable input in a g_i appears in a b_j)

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k^* \left(\mathrm{IF}\ b_k^* \mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \left(\mathop{\mathrm{IF}}\limits_{j=1}^{m} b_k^* \wedge b_j^* \, p_{ij}\right)\right)$$

(by <$\wedge$ - IF distrib>(1.8) and <IF assoc>(1.1))

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k^* \left(\mathrm{IF}\ b_k^* \mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \ \mathrm{IF}\ b_k^* \, p_{ik}\right)$$

(by <IF - <u>false</u> unit>(1.5) and <IF - sym>(1.3) since $b_k^* \wedge b_j^* \equiv \underline{false}$ when $j \neq k$)

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} b_k^* \left(\mathrm{IF}\ b_k^* \left(\mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \, p_{ik}\right)\right)$$

(by <IF - ALT distrib>(2.10))

$$= \mathop{\mathrm{IF}}\limits_{k=1}^{m} \mathop{\mathrm{ALT}}\limits_{i=1}^{n} g_i \, p_{ik} \qquad \text{(by the lemma)}$$

3. <u>The normal form</u>

We cannot claim that IF/ALT is a normal form since even though it has a far more restricted syntax than general occam, it is still possible to have equivalent programs with essentially different syntax. This is because its construction did not take account of many of the equivalences that can arise between IF constructs, between ALT constructs, or as a consequence of <IF – ALT distrib> (2.10), the law which relates the two. The following examples illustrate some non-trivial forms of equivalence that are not recognised by reduction to IF/ALT. After each example we indicate the way in which our normal form will solve the problem illustrated.

a) It is possible to have clauses in IF constructs that are never executed, because the associated booleans must always evaluate to false. Some such cases are obvious, as when <u>false</u> is itself one of the booleans, but some are more subtle, as in

```
IF                               IF
  x mod 2 = 1                      x mod 2 = 1
    IF              =                Q
      x = 0
        P
      x ≠ 0
        Q
```

where, in the lefthand process, one of the booleans in the inner IF is always false because of its context.

In the normal form all such clauses will be eliminated from conditionals by using <IF – <u>false</u> unit> (1.5). Difficulties such as those posed by the above example will be avoided by making sure that any boolean appearing within the "scope" of another is stronger than it.

The above example also illustrates the point that if, in $\mathrm{IF}_{i=1}^{n}\ b_i\ P_i$, any of P_i is a conditional, then it may be unfolded using $\langle \wedge-\mathrm{IF}$ distrib$\rangle$ (1.6), etc. The normal form never has one IF directly as the argument of another.

b) It is sometimes possible to make a conditional choice before it is strictly required, and always possible to introduce a meaningless choice (between two identical processes). Consider the process

```
IF
  x = 0
    ALT(c!1 P)
  x > 0
    ALT(c!0 Q)
  x < 0
    STOP   .
```

This has essentially different beheviours depending on $x \geqslant 0$ or $x < 0$ (it either can communicate or not): this conditional choice is therefore unavoidable. On the other hand, the choice between $x = 0$ and $x > 0$ can be postponed to (at least) the next step: it is only the value communicated down c that is at stake, and it is possible to construct a single expression that takes the correct value in all states with $x \geqslant 0$. If b,e,f are expressions, we will use the notation $e \nless b \ngtr f$ for the expression that takes value e if b is "true" and f if b is "false". (We do not specify its value for other values of b.) The program above may be transformed to

```
IF
  x ≥ 0
    ALT(c!(1 ≮ x = 0 ≯ 0)  IF(x = 0 P, x ≥ 0 Q))
  x < 0
    STOP
```

by a combination of substitution of expressions, <IF sym> (1.3), <∧ - IF distrib> (1.8) and <ALT - IF distrib> (the derived law proved at the end of Section 2).

In our normal form only strictly necessary choices will be made, and these will be made as late as possible.

c) There are several ways in which apparently different ALT constructs can give the same effect. For example,

```
ALT
  c?x
    P                              and          ALT(c?x P, G)

  SKIP
    ALT(c?x P, G)
```

are equivalent.

If the communication option of the first process <u>is</u> taken up, the environment cannot tell it is not operating the second (for exactly the same option is present there). If that option is not offered or not taken up, the first process quickly transforms itself (by the operation of the SKIP guard) to the second.

The above equality cannot be proved from our existing laws, since (as we have already stated) the laws of ALT are not yet complete. We will shortly develop the further laws needed to counter this type of equivalence.

d) If, at some point, a program can output several different expressions on the same channel, or assign several different expressions to the same variable, some subtle difficulties appear. (Such behaviour can easily arise in occam because of nondeterminism.) A pair of expressions may, as the state varies, sometimes evaluate to the same value and sometimes to different values. For example

```
ALT
  c!0
    P
  c!(x mod 2)
    Q
```

is clearly equivalent to

```
IF
  (x mod 2) = 0
     ALT
       c!0
         ALT
           SKIP
               P
           SKIP
               Q

  (x mod 2) ≠ 0
    ALT
      c!0
        P
      c!1
        Q
```

since, if (x mod 2) = 0, communicating 0 can lead down either branch of the first program.

In our normal form we will insist that if two expressions are both available as outputs on the same channel, or for assignment to the same variable, then they <u>are</u> different. (In no state where they are evaluated do they take the same value.)

Even this restriction is not enough: consider the following pair of processes.

```
ALT                          ALT
  SKIP                         SKIP
    x:=0                         x:=x mod 2
  SKIP                         SKIP
    x:=1                         x:=1 - (x mod 2)
```

They are clearly equivalent, even though there is no one-to-one matching between the pairs of expressions that appear in them. Just because, in every state, the sets $\{0,1\}$ and $\{x \bmod 2,\ 1-(x \bmod 2)\}$ are the same, does not mean that there is any uniform equivalence between the individual expressions. In the normal form we are forced to accept only one of these representations; we choose the left hand one by insisting that pairs of expressions $\{e_1,e_2\}$ output on the same channel or assigned to the same variable be ordered. This means that in all states where they are evaluated, e_1 (say) is always strictly larger than e_2. (The linear order chosen is of little consequence, provided it is expressible in the language. We will assume the identification of all possible expression values with distinct integers.)

For a convincing construction of a normal form it is not enough merely to list a few types of equivalence that can arise and show how to deal with them. This approach can never tell us that there are no more (even more subtle) equivalences waiting to be discovered. Instead we must construct a normal form explicitly around the semantic properties of programs: it should be obvious that different normal form programs are different semantically. A good example is "full disjunctive normal form" for propositional formulae. There is an obvious and close correspondence between the syntax of full d.n.f. formulae and the underlying semantics (functions from truth assignments to $\{true,false\}$).

An occam process can be thought of as acting in steps: a step is either a single communication or the act of successful termination.

The normal form will characterise the first step of a process' behaviour using the highest levels of syntax, and rely on inner levels to deal with subsequent steps. There are three essentially different ways in which the first step can be influenced.

(i) It can depend on the values of the program's variables. This type of choice is typified by IF constructs.

(ii) It can depend on internal decisions by the process that are nondeterministic and invisible to the environment. The purest form of this is in ALT constructs with SKIP guards: for example ALT(SKIP P, SKIP Q) is a process that is free to behave like P or like Q, the choice depending neither on the environment nor on the program's variables.

(iii) An occam process can offer its environment a choice of communications: its first step behaviour then depends on the choice made by the environment. This choice might be at the level of choosing what to output to the process along a particular channel, or of choosing (via an ALT with communication guards) which channel to communicate on.

To describe a process' first step behaviour we will thus use three levels of syntax: essentially one for each variety of choice.

The normal form has two parameters. The first is a boolean expression representing all facts known about the process' free variables. This is necessary because, as was shown in example (a) above, it is necessary to take account at inner levels of conditionals already passed through. The other parameter, inherited from IF/ALT, is a list of free variables.

To keep our individual definitions as simple as possible we will define two sorts of program mutually. A $b,\underset{\sim}{x}$- normal form program has conditional choice (type (i) above) at its outermost level, while a $b,\underset{\sim}{x}$- ALT pattern has a mixture of the other two.

Definition A $b,\underset{\sim}{x}$- normal form is a program of the form

$$\underset{i=1}{\overset{n}{\mathrm{IF}}}\ b_i\ P_i,$$

where the b_i partition b, for no i is $b_i \equiv$ false, and the P_i are distinct $b_i,\underset{\sim}{x}$- ALT patterns.

(ALT patterns, perhaps with different boolean parameters, are distinct if they cannot be reconciled to a single choice, as was done in example (b) above. A formal definition of this notion will be supplied later.)

An ALT pattern will be a way of characterising the behaviour of a process whose general shape of first-step behaviour is the same for all permitted initial values of its free variables. This "shape" is determined by looking at the range of first step behaviours open to the process.

There are four essentially different things a process can do on its first step:

(i) it diverges;

(ii) it communicates with its environment (and goes on to its second step);

(iii) it stops because, even though it has not terminated, it cannot agree with its environment on any communication;

(iv) it terminates in some state.

The "shape" of a process' first step will be a mixture of possibilities from the above. Nondeterminism within the process, and the many choices open to the environment, mean that any mixture of these containing at least one of $\{i, iii, iv\}$ is possible. (It is impossible to construct a process that communicates in every circumstance. This is because any process can be faced with an environment that will not agree to any communication.) Recall, however, that we havs chosen to identify all processes that can diverge. Thus $\perp$ will be a $b,\underset{\sim}{x}$-ALT pattern, and all others will be divergence-free on their first steps.

The other $b,\underset{\sim}{x}$-ALT patterns are essentially just lists of the possibls combinations from (ii), (iii) and (iv) above.

<u>Definition</u> The program P is a $b,\underset{\sim}{x}$-ALT pattern iff it is <u>either</u> $\perp$ <u>or</u>

$$\text{VAR } y_1, \ldots, y_n : \underset{i=1}{\overset{N}{\text{ALT}}}\ g_i\ P_i$$

where there are integers K, L with $0 \leq K \leq L \leq N$ and $K < N$ such that

$1 \leq i \leq K$ implies that g_i has one of the forms $c?y_j$ and $c!e$, and that P_i is a $b,\underset{\sim}{x}$-normal form. All input channels are distinct, and the (distinct) variables used in input guards are precisely $y_1, \ldots, y_n$ (none of which is a component of $\underset{\sim}{x}$). y_j is not free in $g_i\ P_i$ unless $g_i = c?y_j$. If $c!e$ and $c!f$ are two different g_i then $b \models e < f$ or $b \models f < e$. For each i, bound(P_i) is disjoint from free(P), $\{y_1, \ldots, y_n\}$ and the components of $\underset{\sim}{x}$.

$K < i \leq L$ implies g_i is SKIP and P_i is $\underset{j \in X_i}{\text{ALT}}\ g_j\ P_j$ where the X_i $(K < i \leq L)$ are incomparable subsets of $\{1, \ldots, K\}$ with the property that if $g_r = c!e$ and $g_s = c!f$ (both outputs on the same channel), then $s \in X_i \iff r \in X_i$. (The sets X and Y are said to be incomparable if $X \nsubseteq Y$ and $Y \nsubseteq X$.)

$L < i \leq N$ implies g_i is SKIP and P_i is $\underset{\sim}{x}:=\underset{\sim}{e}_i$ where, if e_{ij} denotes the jth expression in the vector $\underset{\sim}{e}_i$, we always have $b \models e_{ij} = e_{kj}$ or $b \models e_{ij} > e_{kj}$ or $b \models e_{ij} < e_{kj}$. Furthermore, if $i \neq k$, there exists some j with $b \models e_{ij} \neq e_{kj}$.

Clearly the first K guards correspond to the process' possible communications, the next L-K to the minimal combinations of communications it can choose to accept from (but not terminate), and the final N-L to its possible final states (after termination). The condition $K < N$ asserts that the process must be able either to terminate or to stop.

The reasons for demanding that expressions output on one channel, or assigned to the same variable, be uniformly ordered have already been explained. Most of the other constructions should be reasonably clear except possibly the construction of the section $K < i \leq L$.

This section is present to identify those environments with which the process might deadlock (i.e. stop because it cannot agree any communication with the environment). Observe that the process is free to execute any of the corresponding SKIP guards (g_i for $i \in \{K+1, \ldots, L\}$ and can only deadlock if it does execute one of these guards. Thus deadlock can occur if and only if the environment offers to communicate on a set of channels disjoint from one of the sets represented by the P_i $(K < i \leq L)$.

It is clear that the set of such environments would not be changed by introducing an additional option with a larger set of P's communications than one of the P_i $(K < i \leq L)$, because whenever it can deadlock, so can P_i. This is why we only record minimal acceptances, or in other words, why we insist that the X_i $(K < i \leq L)$ are incomparable.

On the other hand processes with different sets of minimal acceptances are observably different. This is clear when we note that, given two different collections of incomparable subsets of $\{1, \ldots, K\}$, one must contain an element X that is not a superset of any element of the other. Thus there is a set of channels (the complement of those represented by X) that the environment can offer which one process can deadlock on but not the other.

Note that the whole set $\{g_1, \ldots, g_k\}$ or the empty set can appear as minimal acceptances, but that if one of them does appear then it is the only minimal acceptance (i.e. $L = K+1$). The first of these happens when the process can fail to terminate but there is no communication it can either accept or refuse. The second occurs when the process has the option of deadlocking completely: getting into a nonterminated state where no communication is possible.

All outputs along the same channel always appear together in the minimal acceptances because we assume that the environment, like occam processes, does not have the power of selective input on a channel. Thus we do not discriminate between a process that offers to output one of two values on a channel nondeterministically and one that offers the choice to the environment, even if this last idea were operationally reasonable. No environment we allow is equipped to observe such distinctions. The minimal acceptances are thus essentially sets of channels, and so in constructing them we must identify all guards corresponding to the same channel. (This problem does not arise with input channels because these are all, by assumption, distinct in ALT patterns.)

The list of communications ($1 \leqslant i < K$) needs to be represented independently of the minimal acceptances because not all communications need appear in a minimal acceptance set. Indeed, it is possible to have communications but no minimal acceptances at all, as in ALT(c?x SKIP, SKIP SKIP).

Notice that each communication guard g_i is always followed by the same process P_i, whether it appears in the communication section or the minimal acceptances section. This is because our semantic model (chosen because it expresses the weakest equivalence required for most practical correctness issues) does not distinguish between processes on the grounds of what communications can be observed after the refusal of specific sets. For example, we regard the two processes

a)
```
ALT
  SKIP
    ALT
      c?x
        c?x
      d?x
        c?x
  SKIP
    ALT
      c?x
        STOP
```

and b)
```
ALT
  SKIP
    ALT
      c?x
        c?x
      d?x
        c?x
  SKIP
    ALT
      c?x
        STOP
      c?x
        c?x
```

as equivalent, even though they have different possible behaviours once the refusal of "d" has been observed and an input has been made on channel c.

A finer model (i.e. one identifying less processes) might necessitate different processes after different instances of a guard. It might also be necessary to include more acceptances than just the minimal ones in order to accommodate this type of distinction.

We can extract from each $b,\underset{\sim}{x}$ - ALT pattern an <u>abstract shape</u> for the behaviour it represents. It is either $\perp$ or a triple, whose first component is a set of directed channels, the output channels having a multiplicity. Its second component is a set of incomparable subsets of the channels. The final component is a set of k-tuples of positive integers, where k is the length of $\underset{\sim}{x}$. For each $i \in \{1, \ldots, k\}$ the set of ith components of these tuples has the form $\{1, 2, \ldots, n_i\}$ for some $n_i \geq 0$. For example, if $\underset{\sim}{x} = \langle x_1, \ldots, x_k \rangle$ the tuple $\langle 1, 3, \ldots, 2 \rangle$ means "assign the smallest of x_1's expressions to it, the third smallest of x_2's expressions to it, ..., and the second smallest of x_k's expressions to it". Note that the second and third components of the triple cannot both be empty.

Recall that the $b_i,\underset{\sim}{x}$ - ALT patterns P_i making up the normal form program $\underset{i=1}{\overset{n}{IF}}\, b_i\ P_i$ must be distinct, in that for no i and j can $IF(b_i\ P_i,\ b_j\ P_j)$ be transformed into a $b_i \vee b_j,\underset{\sim}{x}$ - ALT pattern. We define ALT patterns to be <u>distinct</u> if they have different abstrect shapes. Note that this corresponds well to our objective of having the outar conditional in the normal form determine the shape of first step behaviour. It is easy to see that two non- $\perp$ ALT patterns fail to be distinct if and only if there are straightforward permutations of the communications, minimal acceptances and terminations of the first that match the second (except for names of input variables and the various expressions, but preserving order of expressions). If such a set of permutations exists we will call them a <u>matching</u> of the two ALT patterns.

<u>Definition</u>

Let $P = VAR\ x_1, \ldots, x_m : \underset{i=1}{\overset{N}{ALT}}\, g_i\ P_i$

with $K < i \leq L \Longrightarrow g_i = SKIP$ and $P_i = \underset{j \in X_i}{ALT}\, g_j\ P_j$

and $L < i \leq N \Longrightarrow g_i = SKIP$ and $P_i = \underset{\sim}{x} := \underset{\sim}{e}_i$

and $Q = \text{VAR } y_1, \ldots, y_{m^*} : \underset{i=1}{\overset{N^*}{\text{ALT}}}\ h_i\ Q_i$

$$\text{with } K^* < i \leq L^* \implies h_i = \text{SKIP and } Q_i = \underset{j \in Y_i}{\text{ALT}}\ h_j\ Q_j$$

$$\text{and } L^* < i \leq N^* \implies h_i = \text{SKIP and } Q_i = \underset{\sim}{x} := \underset{\sim}{f}_i$$

be respectively b and $b^*, \underset{\sim}{x}$ - ALT patterns. If $N = N^*$, $m = m^*$, $K = K^*$ and $L = L^*$ a <u>matching</u> of P and Q is a quadruple $< \nu, \gamma, \rho, \mathcal{T} >$ of bijections $\nu : \{1, \ldots, m\} \longrightarrow \{1, \ldots, m\}$, $\gamma : \{1, \ldots, K\} \longrightarrow \{1, \ldots, K\}$, $\rho : \{K+1, \ldots, L\} \longrightarrow \{K+1, \ldots, L\}$, $\mathcal{T} : \{L+1, \ldots, N\} \rightarrow \{L+1, \ldots, N\}$ such that

a) if $g_i = c?x_j$ then $h_{\gamma(i)} = c?y_{\nu(j)}$;

if $g_i = c!e$ then $h_{\gamma(i)} = c!e^*$ for some e^* ;

if $g_i = c!e$, $g_j = c!f$, $h_{\gamma(i)} = c!e^*$

and $h_{\gamma(j)} = c!f^*$, then $b \models e < f$ iff $b^* \models e^* < f^*$.

b) $Y_{\rho(i)} = \{ \gamma(j) \mid j \in X_i \}$

c) if the jth components of $\underset{\sim}{e}_i$ and $\underset{\sim}{f}_i$ are respectively denoted e_{ij} and f_{ij} then

$$b \models e_{ij} < e_{kj} \iff b^* \models f_{\mathcal{T}(i)j} < f_{\mathcal{T}(k)j}$$

$$b \models e_{ij} = e_{kj} \iff b^* \models f_{\mathcal{T}(i)j} = f_{\mathcal{T}(k)j}$$

$$b \models e_{ij} > e_{kj} \iff b^* \models f_{\mathcal{T}(i)j} > f_{\mathcal{T}(k)j} .$$

This completes our definition of the normal form. Our objective when constructing the normal form was that two such programs would only be semantically equivalent if they were syntactically equivalent in some obvious way. There are three ways in which two $b, \underset{\sim}{x}$ - normal form programs can be semantically equivalent.

(i) The operators ALT and IF (with disjoint booleans) are symmetric. Thus their arguments can be permuted without changing the semantics of a normal form program.

(ii) The names of bound variables may be changed.

(iii) Any expression can be replaced by another one which is equivalent. In the case of expressions output on channels or assigned to variables this expression only needs to hold in the context of the strongest enclosing boolean.

Programs that are equivalent for reasons (i) and (ii) above are readily proved equivalent using the laws. Programs that are equivalent for the third reason are proved equivalent by the following rule.

Rule of substitution for expressions

a) If e is any expression appearing in the program P and $\models$ e = e', then provided P', a program in which some occurrence of e has been replaced by e', is correct, P = P'.

b) If b $\models$ e = e' then IF b ALT(c!e P, $\underset{\sim}{G}$) = IF b ALT(c!e' P, $\underset{\sim}{G}$).

c) If b $\models$ e = e' then IF b x:=e = IF b x:=e'.

In fact (i), (ii) and (iii) (and combinations thereof) are the only ways in which a pair of b,$\underset{\sim}{x}$-normal form programs can be semantically equivalent. We thus formally define equivalence of normal forms as follows.

<u>Definition</u> a) The $b,\underset{\sim}{x}$-normal form programs $\overset{n}{\underset{i=1}{IF}}\ b_i\ P_i$ and $\overset{n'}{\underset{i=1}{IF}}\ b'_i\ P'_i$ are <u>equivalent</u> if and only if $n = n'$ and there is a bijection $\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ such that, for each i, $\models b_i = b'_{\sigma(i)}$ and P_i is equivalent (as an ALT pattern) to $P_{\sigma(i)}$.

b) The $b,\underset{\sim}{x}$ ALT patterns P and Q are <u>equivalent</u> if and only if <u>either</u> they are both $\perp$, <u>or</u>

$$P = \text{VAR } x_1, \ldots, x_n : \overset{N}{\underset{i=1}{ALT}}\ g_i\ P_i$$

with $K < i \leq L \implies g_i = \text{SKIP}$ and $P_i = \underset{j \in X_i}{ALT}\ g_j\ P_j$

and $L < i \leq N \implies g_i = \text{SKIP}$ and $P_i = \underset{\sim}{x} := \underset{\sim}{e}_i$

$$Q = \text{VAR } y_1, \ldots, y_n : \overset{N}{\underset{i=1}{ALT}}\ h_i\ Q_i$$

with $K < i \leq L \implies h_i = \text{SKIP}$ and $Q_i = \underset{j \in Y_i}{ALT}\ h_j\ Q_j$

and $L < i \leq N \implies h_i = \text{SKIP}$ and $Q_i = \underset{\sim}{x} := \underset{\sim}{f}_i$

and there is a matching $(\vartheta, \gamma, \rho, \mathcal{J})$ between them such that $b \models e = f$ whenever e (from P) end f (from Q) appear "at the same point" (i.e. $g_i = c!e$ and $h_{\gamma(i)} = c!f$, or $e = e_{ij}$ and $f = f_{\mathcal{J}(i)j}$) and such that $1 \leq i \leq K$ implies that P_i is equivalent to $Q_{\gamma(i)}\left[\langle x_1, \ldots, x_n\rangle / \langle y_{\vartheta(1)}, \ldots, y_{\vartheta(n)}\rangle\right]$ as a $b,\underset{\sim}{x}$-normal form.

<u>Theorem 2</u>

The $b,\underset{\sim}{x}$-normal form programs P and Q have IF b P and IF b Q semantically equivalent in the sense of [R] if and only if they are equivalent.

We cannot give a detailed proof of this important result here since it depends so crucially on the details of the denotational semantics, which have not been described in this paper. The following is an outline of the proof of the "only if" part. (The "if" part being much easier.)

So suppose $P = \underset{i=1}{\overset{n}{IF}}\ b_i\ P_i$, $Q = \underset{i=1}{\overset{n'}{IF}}\ b'_i\ Q_i$ and IF b P and IF b Q are semantically equivalent. It is possible to recover the abstract shape of a process' first step behaviour from its semantics. Hence for every state satisfying b, P and Q must have identical shapes of first step behaviour. Now the distinctness of the ALT patterns making up P and Q means that the sets of booleans $\{b_1, \ldots, b_n\}$ and $\{b'_1, \ldots, b'_{n'}\}$ both partition the states satisfying b according to these shapes. From this we can deduce that $n = n'$ and that there is a bijection $\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ such that for each $1 \leqslant i \leqslant n$, $i \models b_i = b'_{\sigma(i)}$ and either $P_i = Q_{\sigma(i)} = \perp$ or there is a matching between P_i and $Q_{\sigma(i)}$. In the latter case it is easily shown that the matching in fact yields an equivalence once induction has been used to deal with lower levels.

Three more laws

There is an important gap that needs to be filled: the last three laws of ALT. They all concern SKIP guards in ALT constructs: the situation where the process is given an option that it can choose invisibly and automatically. In particular, they show what sort of equivalences arise between the type of nondeterministic processes these give rise to. In studying these laws the reader should bear in mind our philosophy that nondivergent processes are equivalent if they have the same communications, minimal acceptances and terminations, and if their possible behaviours after each communication are equivalent. These laws more than any others depend on the way our semantic model treats non-determinism, and would probably need to be revised in other systems.

The firet law says that if the process communicates, the environment is not intsrested in whether this occurred before or after a SKIP guard.

(2.11) $$ALT(SKIP\ ALT(g_1\ P,\ \underset{\sim}{G}_1),\ g_2\ Q,\ \underset{\sim}{G}_2)$$

$$=\ ALT(SKIP\ ALT(g_1\ P,\ g_2\ Q,\ \underset{\sim}{G}_1),\ \underset{\sim}{G}_2)$$

provided <u>either</u> g_1 = c?x and g_2 = c?y

<u>or</u> g_1 = c!e and g_2 = c!f <ALT – SKIP sym>

The fact that the process on the left hand side has a communication on the same channel as g_2 within the inner ALT ensures that both processes have the sams minimal acceptances. The fact that, in the case g_1 = c!e and g_2 = c!f, e need not equal f, expresses the fact that the environment is not capable of inputting selectively on channel c.

The second law allows us to aliminate nested ALTs with SKIP guards. It says that if an ALT can SKIP to a second ALT, which in turn can SKIP to P, then all other options in these ALTs are in exactly the same position: they might be offered, or might be ignored in favour af P.

(2.12) $$ALT(SKIP\ ALT(SKIP\ P,\ \underset{\sim}{G}_1),\ \underset{\sim}{G}_2) = ALT(SKIP\ P,\ \underset{\sim}{G}_1,\ \underset{\sim}{G}_2)$$

<ALT – SKIP reduction>

The final law depends on the fact that we are only interested in <u>minimal</u> acceptance sets. Thus the following two processes with the same communication options (and subsequent behaviours) are equivalent:

(2.13) $$ALT(SKIP\ ALT(\underset{\sim}{G}_1),\ SKIP\ ALT(\underset{\sim}{G}_1,\ \underset{\sim}{G}_2),\ \underset{\sim}{G}_3)$$

$$= ALT(SKIP\ ALT(\underset{\sim}{G}_1),\ \underset{\sim}{G}_2,\ \underset{\sim}{G}_3)$$ <convexity>

The left hand process can SKIP to two options, one of which is a subset of the other. If one of the lists $\underset{\sim}{G}_1$ and $\underset{\sim}{G}_2$ contains a SKIP guard the equivalence is quite easy to see. If neither does it is clear that both processes have exactly the same possible communications, and furthermore any environment which can deadlock with either can deadlock with SKIP ALT($\underset{\sim}{G}_1$) or some SKIP option within $\underset{\sim}{G}_3$.

We now have enough laws to completely capture the semantics of our version of occam. There is one exception: the case of uninitialised variables. The nondeterminism introduced by these is of a particularly difficult kind. Given that any instance of one of these is erroneous, it is not worth putting a great deal of effort into their study. Any use of such a variable by a program will show up in its IF – ALT form. We will thus not attempt to transform any further an IF – ALT program with the "uninitialised variable" construct within it. (Notice that we have not included the possibility of uninitialised variables within normal form programs, since no bound variable is ever read until it has been input to.)

Given Theorem 2 above, the following theorem shows that we have achieved our objective of completely characterising the semantics of finite programs.

<u>Theorem 3</u> If the list $\underset{\sim}{x}$ contains every free variable that the finite program P ever inputs or assigns to, and if P never evaluates an uninitialised variable, then there is a <u>true</u>, $\underset{\sim}{x}$ – normal form program P' such that $\text{free}(P') \subseteq \text{free}(P) \cup \underset{\sim}{x}$ and P = P' is provable from our laws and the rule of substitution for expressions.

By virtue of Theorem 1 it is sufficient to prove this for the case when P is an $\underset{\sim}{x}$ – IF/ALT program.

The proof of Theorem 3 takes very much the same form as that of Theorem 1: it is a recursive procedure for transforming IF b P to $b,\underset{\sim}{x}$-normal form, where P is an $\underset{\sim}{x}$-IF/ALT program without uninitialised variables. Indeed in some ways the proof is rather simpler than Theorem 1, since it does not need such a complex structure of nested recursions. (The reason for this is that IF/ALT and normal form share the property that syntactic structure corresponds closely to execution order: things at high syntactic levels are executed first.)

Theorems 2 and 3 together give us a relative completeness result: relative to the knowledge we are assuming about expressions, our algebraic laws are complete with respect to deciding the equivalence of finite programs. Recall the relation $P \sqsubseteq Q$ introduced in the second section, meaning "Q is more deterministic than P". This was formally defined

$$P \sqsubseteq Q \;\equiv\; P = \text{ALT}(\text{SKIP } P,\ \text{SKIP } Q).$$

It is therefore (relatively) decidable for finite programs using our laws.

It is a fact that, provided the set of "basic values" that expressions can take is finite, the finite programs are finite in the lattice-theoretic sense of the word. In other words, if D is a directed set of processes (under $\sqsupseteq$), P is finite and $\bigsqcup D \sqsupseteq P$, then there is some $Q \in D$ such that $Q \sqsupseteq P$. Thus the following theorem is an easy corollary to Lemma 3.

<u>Theorem 4</u> If P and Q are two occam programs with the property

$$(*) \quad \forall \ulcorner P' \urcorner \in \text{FIN}(\ulcorner P \urcorner).\ \exists \ulcorner Q' \urcorner \in \text{FIN}(\ulcorner Q \urcorner).\ P' \sqsubseteq Q'$$

then $P \sqsubseteq Q$. If the underlying set of basic values is finite, $(*)$ holds if and only if $P \sqsubseteq Q$.

Since $P = Q$ is equivalent to $P \sqsubseteq Q$ and $P \sqsupseteq Q$, Theorem 4 proves the soundness and, in the finite set of values case, completeness of the following infinitary rule for deciding equivalence.

<u>Infinitary rule 1</u> Suppose P and Q are such that

$$\forall \ulcorner P' \urcorner \in FIN(\ulcorner P \urcorner). \exists \ulcorner Q' \urcorner \in FIN(\ulcorner Q \urcorner). \; P' \sqsubseteq Q'$$

and $\forall \ulcorner Q' \urcorner \in FIN(\ulcorner Q \urcorner). \exists \ulcorner P' \urcorner \in FIN(\ulcorner P \urcorner). \; Q' \sqsubseteq P'$

then we may infer $P = Q$.

This rule, together with our laws and the rule of substitution for expressions is enough to completely characterise the semantics of occam if the set of values is finite.

Our use of an infinitary rule, which requires an apparently infinite amount of work to verify its preconditions, appears undesirable. Indeed for any particular finite value set it will be possible to give a complete finitary rule based on the fact that, since any program only contains finitely many variables, it can be regarded as a finite state machine (with a huge number of states). However any such rule would be inelegant and be impossible to apply in practice because of the prohibitive amount of case checking required. Indeed our infinitary rule may well be more practical, since it will be possible to verify its preconditions by induction in many applications.

It should be noted that there is no chance of a complete finitary rule when the value space is infinite. For example we could take our value space to be the integers (with the truth values embedded somehow). We restrict the language of expressions to the comparison and boolean operations (including $\nless \ngtr$ - see Example b of this section), + and -. This means that the facts $b_1 \models b_2$ we are assuming are in principle decidable,[1] and so add nothing to the real power of our system. A complete finitary rule for this language would allow us to decide the halting of arbitrary register machine programs: this is well-known to be impossible. (We have taken care here to ensure that an unscrupulous user could not make use of the calculus of expressions to reason about the large scale structure of programs. It would of course be completely outside the spirit of our style of proof system for him ever to do this.)

Unfortunately Infinitary rule 1 as it stands is not strong enough to give us a complete system when the set of basic values is infinite. Suppose the value space is the integers, and consider the following pair of programs.

```
IF                          WHILE y≠0
  y≥0                         SEQ
    SEQ          and            y:=y - 1
      x:=x + y                  x:=x + 1
      y:=0
  y<0
    ⊥
```

These are equivalent, but the rule does not prove this because the left hand program is finite but is not weaker than any finite syntactic approximation to the right hand program. This is because, as the initial state varies, the number of iterations of the WHILE loop varies unboundedly.

There are several methods of extending our rule to cope with this problem, all of which are essentially ways of considering programs restricted so that we only need worry about a finite set of values at a time.

---

1. The theory of these expressions reduces to that of Presburger arithmetic (see, for example, [F]).

It is quite easy to restrict normal form programs to finite sets of values. Given any list of variables $\underset{\leftrightarrow}{y}$ and finite set of constant expressions[2] F, it is easy to construct a boolean $b^F_{\underset{\sim}{y}}$ which is true if and only if every element of $\underset{\leftrightarrow}{y}$ is in F. All we have to do is to introduce extra conditions of the form $b^F_{\underset{\sim}{y}}$ into the conditionals of the normal form, with an "escape" clause of $\perp$.

Definition

a) If $P = \underset{i=1}{\overset{n}{IF}}\; b_i \; P_i$ is a $b,\underset{\leftrightarrow}{x}$- normal form program and F is a finite set of constant expressions we define $P\downarrow F$ to be

$$IF(\neg b^F_{\underset{\sim}{y}} \perp, (b^F_{\underset{\sim}{y}} \wedge b_1)\; P_1\downarrow F, \; \ldots, (b^F_{\underset{\sim}{y}} \wedge b_n)\; P_n\downarrow F)$$

where $\underset{\leftrightarrow}{y}$ is the list of all variables appearing free in P.

b) If $P = \underset{i=1}{\overset{n}{ALT}}\; g_i \; P_i$ is a $b,\underset{\leftrightarrow}{x}$- ALT pattern and F is a finite set of constant expressions we define $P\downarrow F$ to be the program in which $\downarrow F$ is applied to each normal form appearing after a communication or within a minimal acceptance.

(Note that $P\downarrow F$ need not be a normal form program if P is, since the clauses in the IF^s might be false or not all distinct.)

The following lemma expresses the important properties of the $P\downarrow F$.

Lemma 4 Suppose P is a normal form program and that every value is expressed by some constant expression, then we have:

a) $\{P\downarrow F \mid F \text{ is a finite set of constant expressions}\}$ is directed (under $\sqsubseteq$) with limit P.

---

2. A constant expression is one which contains no variables.

b) For each F, if D is a directed set of processes with $\bigsqcup D \sqsupseteq P{\downarrow}F$, then there is some $Q \in D$ with $Q \sqsupseteq P{\downarrow}F$.

We can associate a set of these "ultra-finite" programs with each occam program P as follows.

$$\mathcal{F}(P) = \{P'{\downarrow}F \mid F \text{ is a finite set of constant expressions and } P' \text{ is a normal form equivalent of some } P'' \in FIN(P)\}.$$

Lemmas 3 and 4 now combine to prove the soundness and completeness of the following rule.

<u>Infinitary rule 2</u> Suppose the programs P and Q are such that

$$\forall P' \in \mathcal{F}(P). \exists\, Q' \in \mathcal{F}(Q).\ P' \sqsubseteq Q'$$

and $\forall Q' \in \mathcal{F}(Q). \exists\, P' \in \mathcal{F}(P).\ Q' \sqsubseteq P'$

then $P = Q$.

We have now completed our characterisation of the semantics of occam. The algebraic laws, infinitary rule 2 and the rule of substitution in expressions provide a sound and complete system for deciding the equivalence of programs. Unfortunately, infinitary rule 2 is likely to be much harder to use in practice than infinitary rule 1. The facts that it relies on transformation to normal form and uses two separate types of approximation mean that its hypotheses will be much harder to prove by induction than those of the earlier rule. There may be alternative rules that are not so problamatic; in particular it should be possible to eliminate the need to transform every program to normal form. This is a topic for future research.

4. Conclusions and prospects

In the first section of this paper we saw that algebraic laws provide a novel but precise framework for describing and defining occam. The completeness of this description was shown by the rest of the paper. This approach can also be used to good effect with other well constructed languages: this is illustrated in [laws], where a simple sequential language (Dijkstra's language of guarded commands [D]) is considered.

The algebraic approach to programming language semantics has several features to recommend it. Laws do not require the construction of complex mathematical models. Each group of laws is fairly self contained and usually easy to understand. Thay are very modular: a change which, with denotational semantics, would require alterations to the mathematical model and consequent revision of every semantic clause, may well require the altaration of only one or two laws.

Nevertheless, the algebraic laws can give rise to complex and unexpected interations, leading to a danger that too many programs will be equated. It is therefore desirable to describe the language by an independent semantic technique (for example denotational) and prove that this is congruent to the algebraic semantics. Such a proof will probably follow similar lines to ours: a demonstration that all laws preserve the semantics, the construction of a normal form, and a proof that two different normal form programs have different denotations. Note that in our case it would have been very difficult to construct the normal form without knowing the structure of the denotational model.

Algebraic laws alone only allow us to prove one occam program equal to another. They do not help in proving a program correct with respect to some specification expressed in terms of a more abstract description of its intended behaviour. Correctness proofs might be based on concepts such as satisfaction (sat) [H], the weakest pre-condition [D] or Hoare logic [AFR]. We expect that these methods will be based more usually on the denotational than the algebraic description of occam. However the laws may well be useful for transforming a program after it has been developed, or for making a program more amenable to some proof technique.

We conclude that even though the algebraic and denotational semantics characterise exactly the same equivalence over occam, they are in some sense complementary. Each has a lot to offer to the other.

Nevertheless, there are a number of practical applications for the laws described in this paper: proving programs equivalent to one another, transforming programs to make them more efficient, and transforming programs to a restricted syntax for special applications. In the three following subsections we examine their potential for these applications.

Deciding the equivalence of programs

The most obvious application of the laws is in deciding whether or not a given pair of finite programs are equivalent. Sections 2 and 3 have developed a procedure for doing this. This is a clear candidate for automation. The only parts of this procedure that are not immediately susceptible to practical implementation are those that rely on the assumption of facts about expressions. For some languages of expressions it will be possible in general to decide these facts (though perhaps not very efficiently), and in any reasonable language there should be wide classes of pairs of expressions whose equivalence is decidable. Even in the absence of a complete procedure for deciding expressions it will be possible to automatically transform each finite program to normal form (except perhaps for the inclusion of some false branches in IF statements). In such circumstances the procedure might be able to decide the equivalence of a given pair of programs, and would in all other cases reduce the question of their equivalence to a boolean expression. It might be appropriate to make such a program interactive, allowing it to interrogate its user on difficult facts concerning expressions.

Much of the complexity of the normal form can be attributed to the potential nondeterminism of occam programs. We have seen various ways in which programs can behave unpredictably: the normal form needs enough structure to characterise all of these. In fact transformation to normal form will be an excellent way of analysing the nondeterminism of programs.

In many practical cases the program will be deterministic, in that it cannot diverge and never has any choice over what to communicate or what to assign to its free variables. For these programs, and deterministic sections of others, much of the structure of our normal form will be redundant. If we wish to store and manipulate normal form programs in computers it will be worthwhile investigating this and other topics to discover how they can be made more compact.

A useful system for handling practical program equivalence questions must be able to deal with programs containing loops. Unfortunately, in deciding the equivalence of any pair of programs involving WHILE loops, it is necessary to compare infinitely many pairs of their finite syntactic approximations. As explained in the previous section, any reasonable complete system is bound to be sometimes infinitary.

However it is certain that by extending our set of laws and rules, and by the use of inductive methods, we can develop systems that will require the use of infinitary rules a good deal less often. It is thus likely that we can develop practical finitary proof techniques which are applicable to many pairs of programs involving WHILE.

A typical method would involve attempting to transform programs to some standard form, for example the normal form with the introduction of loops in some tightly defined ways. The incompleteness of such a method would appear either from the impossibility of transforming every program to standard form, or because the standard form was not a true normal form.

For such techniques we will probably need to discover a number of algebraic laws involving WHILE. We have not needed any of these so far, because finite programs contain no loops. Five examples are given below, each of which is easily derived from our existing systems. (Each requires an application of Infinitary rule 1 and induction.)

(W1) WHILE b P = IF(b SEQ(P, WHILE b P), ¬b SKIP) <WHILE expansion>

(W2) WHILE b_1(WHILE b_2 P) = WHILE $b_1 \vee b_2$ IF(b_2 P, <u>true</u> ⊥) <WHILE combination>

(W3) WHILE b P = IF(b WHILE <u>true</u> P, ¬b SKIP)

if no variable appearing in b is input or assigned to by P

<infinite loop>

(W4) WHILE <u>true</u> $\underset{\sim}{x}:=\underset{\sim}{e}'$ = ⊥ <divergent loop>

(W5) WHILE b SEQ(P,Q)

= IF(b SEQ(P, WHILE b SEQ(Q,P),Q), ¬b SKIP)

if no variable appearing in b is input or assigned to in Q

<WHILE reordering>

In addition to laws in this familiar style, it may also be necessary to use more explicitly directed transformations towards particular standard forms. For example the following may be useful if the target is a state – machine like program. Note that an extra variable is introduced as a flag.

```
(w6) WHILE b                VAR x:
       SEQ          =         SEQ
         P                      x:=false
         Q                      WHILE x ∨ b
                                  IF
                                   x
                                    SEQ
                                      Q
                                      x:=false
                                 ¬x
                                    SEQ
                                      P
                                      x:=true
```

if x is not free in the left hand side.

<loop factorisation>

However there is little hope that the above six laws, or any reasonable extension of them, will be adequate for every problem likely to be encountered in practice.

Improving efficiency

The second possible practical application of algebraic laws is for transforming programs to Improve their efficiency in some way. That this is possible reflects the fact that the laws, while preserving all essential abstract correctness properties, do not imply equal efficiency on either side. Occam gives extra scope for this because it is a parallel language: one can improve a program not only by reducing the overall amount of calculation, but also by configuring it for the (possibly parallel) machine on which it is to be run. The second of these objectives may be easier than the first.

In some circumstances one might seek a maximally parallel version of a program, but it is more likely that one will be attempting to optimise it for a particular configuration. This might be a fixed length pipeline, or even a single sequential processor. A typical technique here might be to seek maximally parallel versions of a program, use the symmetry and associative laws of PAR to divide the task into groups of processes suitable for running on single processors in a given network, and then eliminate some of the parallelism within these groups.

A helpful tool for this type of transformation will be a repertoire of laws directly relating sequential and parallel composition. Because these constructions were both eliminated at an early stage of the transformation to normal form, we have so far not needed any such laws. It should also be possible to discover a number of laws which can be used to assist parallelism introduction, for example by making a sequential program more amenable to it, or speeding up the behaviour of a parallel network. A good example of a sequential-to-parallel transformation is provided by the following.

Suppose no two of the processes $P_1, \ldots, P_m$ $(m \geqslant 1)$ can communicate on the same global channel (even internally), that the list $x_1, \ldots, x_n$ $(n \geqslant 1)$ contains each free variable that can be input or assigned to by one P_i and used (in any way) in another, and that no P_i has a free occurrence of any of the channels $c_0, \ldots, c_m$. Then

$SEQ(P_1, \ldots, P_m) =$

$CHAN\ c_o, \ldots, c_m : PAR(U_o:Q,\ U_1:P'_1, \ldots, U_m:P'_m)$

where $Q = SEQ$

```
SEQ
  c_o!x_1
    :
  c_o!x_n
  c_m?x_1
    :
  c_m?x_n
```

and $P'_r =$

```
VAR x_1 ... x_n:
SEQ
  c_{r-1}?x_1
    :
  c_{r-1}?x_n
  P_r
  c_r!x_1
    :
  c_r!x_n
```

U_o claims c_o for output, c_m for input and $x_1 \ldots x_n$ as variables. For $r \in \{1, \ldots, m\}$, U_r claims c_{r-1} for input, c_r for output and all variables and channels used by P_r except $x_1 \ldots x_n$.

This transformation sets up a ring in which the values of the variables shared between the P_i are passed around in sequence. It would be easy to devise a version of this transformation in which the network created was a straightforward pipeline. (This would be in sequence with another simple process for managing the final values of

$x_1 \ldots x_n$.) Note that no P_i can start up until P_{i-1} has terminated: it is this that makes the transformation so general, but it also makes the resulting parallel program useless as it stands. After performing this transformation one would seek to introduce more useful parallelism by transforming the P'_i in ways that remove the temporal dependence between actions in different P'_j. Useful laws for this include <assignment - ALT distrib> (4.6) and simple derived laws such as

$$\mathrm{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ c!f) = \mathrm{SEQ}(c!f\left[\underset{\sim}{e}/\underset{\sim}{x}\right],\ \underset{\sim}{x}:=\underset{\sim}{e})$$ <assignment-output sym>

$$\mathrm{SEQ}(\underset{\sim}{x}:=\underset{\sim}{e},\ c?y) = \mathrm{SEQ}(c?y,\ \underset{\sim}{x}:=\underset{\sim}{e})$$ provided y is not free in $\underset{\sim}{x}:=\underset{\sim}{e}$.

<assignment-input sym>

Unfortunately the corresponding law of input/output symmetry

$$\mathrm{SEQ}(c?x,\ d!e) = \mathrm{SEQ}(d!e,\ c?x)$$

provided x does not appear in e

is <u>never</u> true as it stands. Nevertheless it is a substitution that can be made in a number of contexts where at least one of c and d is used for internal communication.

<u>Transformation to a restricted syntax</u>

The final easily identified practical application for the laws is the transformation of general occam programs into restricted subsets of the language. This paper has shown just how successfully this can be done: we have transformed every finite program to a normal form to which it usually bears no syntactic or structural resemblance. It seems unlikely that the normal form is one into which we would choose to transform programs for execution, but our work gives hope that transformation into other, more useful forms might be tractable.

An important application of this idea is likely to be in VLSI design. Occam is a natural language for specifying and describing systems such as VLSI circuits. The way in which these circuits are built up in a structured way out of interacting modules and submodules corresponds well to the use of nested parallel constructs in occam. In specifying such systems we are likely to use fairly straightforward types of occam, which will make transformation easier. In particular the set of expression values is likely to be much restricted (perhaps allowing only the Boolean values 0 and 1).

Let us suppose that we know that particular types of occam program are directly implementable in silicon by some automated system. Then to implement a circuit specified in occam it will be sufficient to transform it to one of these implementable subsets of occam. Because all our transformations are provably correct, the resulting chip design is guaranteed to be a correct implementation of the original specification.

An essential prerequisite for this work will be the definition of the directly implementable subsets of occam. An obvious candidate is some stylised representation of a finite-state machine. Others will clearly involve parallelism and communication. The handshaken communication of occam can be implemented directly on silicon by asynchronous design rules; and for larger circuits this is an effective method for avoiding problems of clock skew. For smaller circuits with highly regular communications, the occam handshake can sometimes be replaced by a clocked synchronous transfer.

Appendix: A summary of the laws of occam

a) The complete set of laws

Laws of IF

(1.1) $IF(\underset{\sim}{C}_1, IF(\underset{\sim}{C}_2), \underset{\sim}{C}_3) = IF(\underset{\sim}{C}_1, \underset{\sim}{C}_2, \underset{\sim}{C}_3)$ <IF assoc>

(1.2) $\overset{n}{\underset{i=1}{IF}}\ b_i\ P_i = \overset{n}{\underset{i=1}{IF}}\ b_i^*\ P_i$, where $b_i^* = \neg b_1 \wedge \ldots \wedge \neg b_{i-1} \wedge b_i$

<IF priority>

(1.3) $\overset{n}{\underset{i=1}{IF}}\ b_i\ P_i = \overset{n}{\underset{i=1}{IF}}\ b_{\pi(i)}\ P_{\pi(i)}$ for any permutation π of $\{1 \ldots n\}$ provided $b_i \wedge b_j \equiv$ <u>false</u> whenever $i \neq j$

<IF sym>

(1.4) $IF(b_1\ P, b_2\ P, \underset{\sim}{C}) = IF(b_1 \vee b_2\ P, \underset{\sim}{C})$ <IF - $\vee$ distrib>

(1.5)‡ $IF(\underline{false}\ P, \underset{\sim}{C}) = IF(\underset{\sim}{C})$ <IF - <u>false</u> unit>

(1.6)‡ $IF(\underset{\sim}{C}, b\ STOP) = IF(\underset{\sim}{C})$ <IF - STOP unit>

(1.7) $IF(\underline{true}\ P) = P$ <IF - <u>true</u> unit>

(1.8) $IF(\underset{\sim}{C}, b\ \overset{m}{\underset{i=1}{IF}}\ b_i\ P_i) = IF(\underset{\sim}{C}, \overset{m}{\underset{i=1}{IF}}\ b \wedge b_i\ P_i)$ <$\wedge$ - IF distrib>

Laws of ALT

(2.1) $ALT(ALT(\underset{\sim}{G}_1), \underset{\sim}{G}_2) = ALT(\underset{\sim}{G}_1, \underset{\sim}{G}_2)$ <ALT assoc>

(2.2) $\overset{n}{\underset{i=1}{ALT}}\ G_i = \overset{n}{\underset{i=1}{ALT}}\ G_{\pi(i)}$ π any permutation of $\{1 \ldots n\}$

<ALT - sym>

(2.3) $ALT(\ ) = STOP$ <ALT - STOP unit>

(2.4) $ALT(b \,\&\, g\ P, \underline{G}) = IF(b\ ALT(g\ P, \underline{G}), \neg b\ ALT(\underline{G}))$ <boolean guard elim>

(2.5) $ALT(SKIP\ P) = P$ <ALT – SKIP identity>

(2.6) $ALT(c?x\ SKIP) = c?x$ <input>

(2.7) $ALT(c!e\ SKIP) = c!e$ <output>

(2.8) $ALT(g\ P, \underline{G}) = ALT(g\ P, g\ P, \underline{G})$ <ALT idempotence>

(2.9) $ALT(g\ P, g\ Q, \underline{G}) = ALT(g\ ALT(SKIP\ P, SKIP\ Q), \underline{G})$ <guard distrib>

(2.10) $IF\ b \overset{n}{\underset{i=1}{ALT}} g_i\ P_i = IF\ b \overset{n}{\underset{i=1}{ALT}} g_i\ (IF\ b\ P_i)$ provided no variable appearing in be is input in any g; <IF -ALT distrib>

(2.11) $ALT(SKIP\ ALT(g_1\ P, \underline{G}_1), g_2\ Q, \underline{G}_2)$

$= ALT(SKIP\ ALT(g_1\ P, g_2\ Q, \underline{G}_1), \underline{G}_2)$

provided <u>either</u> $g_1 = c?x$ and $g_2 = c?y$

<u>or</u> $g_1 = c!e$ and $g_2 = c!f$ <ALT – SKIP sym>

(2.12) $ALT(SKIP\ ALT(SKIP\ P, \underline{G}_1), \underline{G}_2) = ALT(SKIP\ P, \underline{G}_1, \underline{G}_2)$

<ALT – SKIP reduction>

(2.13) $ALT(SKIP\ ALT(\underline{G}_1), SKIP\ ALT(\underline{G}_1, \underline{G}_2), \underline{G}_3)$

$= ALT(SKIP\ ALT(\underline{G}_1), \underline{G}_2, \underline{G}_3)$ <convexity>

<u>Laws of assignment</u>

(3.1) $\langle\rangle := \langle\rangle = SKIP$ <SKIP>

(3.2) $\langle x_i \mid i = 1 \dots n\rangle := \langle a_i \mid i = 1 \dots n\rangle$

$= \langle x_{\pi(i)} \mid i = 1 \dots n\rangle := \langle e_{\pi(i)} \mid i = 1 \dots n\rangle$

for π any permutation of $\{1 \dots n\}$ <assignment sym>

(3.3)† $\underline{x} + \underline{y} := \underline{e} + \underline{y} = \underline{x} := \underline{e}$ <identity assignment>

Laws of SEQ

(4.1) $SEQ(\ ) = SKIP$ <SEQ – SKIP unit>

(4.2) $SEQ(P, \underset{\sim}{P}) = SEQ(P, SEQ(\underset{\sim}{P}))$ <SEQ assoc>

(4.3)* $SEQ(\overset{n}{\underset{i=1}{IF}}\ b_i\ P_i,\ Q) = \overset{n}{\underset{i=1}{IF}}\ b_i\ SEQ(P_i,\ Q)$ <SEQ – IF distrib>

(4.4)* $SEQ(\overset{n}{\underset{i=1}{ALT}}\ g_i\ P_i,\ Q) = \overset{n}{\underset{i=1}{ALT}}\ g_i\ SEQ(P_i,\ Q)$ <SEQ – ALT distrib>

(4.5)* $SEQ(\underset{\sim}{x} := \underset{\sim}{e},\ \overset{n}{\underset{i=1}{IF}}\ b_i\ P_i) = \overset{n}{\underset{i=1}{IF}}\ b_i \left[{}^{\underset{\sim}{e}}/_{\underset{\sim}{x}}\right] SEQ(\underset{\sim}{x} := \underset{\sim}{e}, P_i)$ <assignment – IF distrib>

(4.6)* $SEQ(\underset{\sim}{x} := \underset{\sim}{e},\ \overset{n}{\underset{i=1}{ALT}}\ g_i\ P_i) = \overset{n}{\underset{i=1}{ALT}}\ g_i \left[{}^{\underset{\sim}{e}}/_{\underset{\sim}{x}}\right] SEQ(\underset{\sim}{x} := \underset{\sim}{e},\ P_i)$

provided no variable which occurs in $\underset{\sim}{x}$ or $\underset{\sim}{e}$ is

<u>input</u> in any g_i. <assignment – ALT distrib>

(4.7)* $SEQ(\underset{\sim}{x} := \underset{\sim}{e},\ \underset{\sim}{x} := \underset{\sim}{f}) = \underset{\sim}{x} := \underset{\sim}{f}\left[{}^{\underset{\sim}{e}}/_{\underset{\sim}{x}}\right]$ <combine assignments>

Laws of PAR

(5.1) $PAR(\ ) = SKIP$ <PAR – SKIP unit>

(5.2) $\overset{n}{\underset{i=1}{PAR}}\ U_i : P_i = PAR(U_1 : P_1,\ U^* : (\overset{n}{\underset{i=2}{PAR}}\ U_i : P_i))$ $(n > 0)$

where U^* is the union of $U_2 \ldots U_n$ <PAR assoc>

(5.3) $PAR(U_1 : P_1,\ U_2 : P_2) = PAR(U_2 : P_2,\ U_1 : P_1)$ <PAR sym>

(5.4)* $PAR(U_1 : \overset{n}{\underset{i=1}{IF}}\ b_i\ P_i,\ U_2 : Q) = \overset{n}{\underset{i=1}{IF}}\ b_i\ PAR(U_1 : P_i,\ U_2 : Q)$

provided $b_1 \vee \ldots \vee b_n \equiv$ <u>true</u> <PAR – IF distrib>

(5.5)* $PAR(U_1 : \underset{\sim}{x} := \underset{\sim}{e},\ U_2 : \underset{\sim}{y} := \underset{\sim}{f}) = \underset{\sim}{x} + \underset{\sim}{y} := \underset{\sim}{e} + \underset{\sim}{f}$ <PAR assignments>

(5.6)‡ ~~If each g_i has one of the forms c?x, c!e or SKIP,~~

then $PAR(U_1:\underset{i=1}{\overset{n}{ALT}}\ g_i\ P_i,\ U_2:\underset{\sim}{x}:=\underset{\sim}{e}) = \underset{i \in X}{ALT}\ g_i\ PAR(U_1:P_i,\ U_2:\underset{\sim}{x}:=\underset{\sim}{e})$

where X is the set of indices $i \in \{1, 2, \ldots, n\}$ such that

g_i = SKIP

or g_i = (b&) c!e and $c \in outs(U_1) - ins(U_2)$

or g_i = (b&) c?x and $c \in ins(U_1) - outs(U_2)$.

<expansion 1>

(5.7)‡ If $P = \underset{i=1}{\overset{n}{ALT}}\ g_i\ P_i$, and $Q = \underset{j=1}{\overset{m}{ALT}}\ h_j\ Q_j$, ~~where each g_i, h_j has one of the forms c?x, c!e or SKIP~~, then $PAR(U_1:P,\ U_2:Q) = \underset{r=1}{\overset{N}{ALT}}\ k_r\ R_r$,

where the pairs $\langle k_r, R_r \rangle$ are precisely all possibilities from the following:

(i) $R_r = PAR(U_1:P_i,\ U_2:Q)$ and

$k_r = g_i$ = SKIP

or $k_r = g_i$ = (b&) c!e and $c \in outs(U_1) - ins(U_2)$

or $k_r = g_i$ = c?x and $c \in ins(U_1) - outs(U_2)$

(ii) $R_r = PAR(U_1:P,\ U_2:Q_j)$ and

$k_r = h_j$ = SKIP

or $k_r = h_j$ = (b&) c!e and $c \in outs(U_2) - ins(U_1)$

or $k_r = h_j$ = c?x and $c \in ins(U_2) - outs(U_1)$

(iii) $R_r = SEQ(x:=e,\ PAR(U_1:P_i,\ U_2:Q_j))$

k_r = ~~SKIP and~~ (b&)(b'&) SKIP (b&) (b'&)

g_i = c!e and h_j = c?x and $c \in ins(U_2) \cap outs(U_1)$

or g_i = c?x and h_j = c!e and $c \in ins(U_1) \cap outs(U_2)$.

<expansion 2>

Laws of declaration

(6.1) $\text{VAR } x_1 : (\text{VAR } x_2 : \ldots \text{VAR } x_n : P) \ldots) = \text{VAR } x_1 \ldots x_n : P$ <VAR assoc>

(6.2) $\text{VAR } x_1 : (\text{VAR } x_2 : P) = \text{VAR } x_2 : (\text{VAR } x_1 : P)$ <VAR sym>

(6.3) $\text{VAR } x : P = P$ if $x \notin \text{free}(P)$ <VAR elim>

(6.4) $\text{VAR } x : P = \text{VAR } y : P\left[{}^{y}/_{x}\right]$ if $y \notin \text{free}(P)$ <VAR rename>

(6.5) $\underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\ (\text{VAR } x : P_i) = \text{VAR } x : (\underset{i=1}{\overset{n}{\text{ALT}}}\ g_i\ P_i)$

provided x is free in no g_i <VAR – ALT distrib>

(6.6) $\underset{i=1}{\overset{n}{\text{IF}}}\ b_i\ (\text{VAR } x : P_i) = \text{VAR } x : (\underset{i=1}{\overset{n}{\text{IF}}}\ b_i\ P_i)$

provided x is free in no b_i <VAR – IF distrib>

(6.7) $\text{SEQ}(\text{VAR } x : P, Q) = \text{VAR } x : \text{SEQ}(P, Q)$ if $x \notin \text{free}(Q)$ <VAR – SEQ 1>

(6.8) $\text{SEQ}(P, \text{VAR } x : Q) = \text{VAR } x : \text{SEQ}(P, Q)$ if $x \notin \text{free}(P)$ <VAR – SEQ 2>

(6.9) $\text{PAR}(U_1 : (\text{VAR } x : P), U_2 : Q) = \text{VAR } x : \text{PAR}(U_1^{\ddagger} : P_1, U_2 : P_2)$,

<u>provided</u> x is not free in $U_2 : P_2$, where $U_1^{\ddagger}$ is U_1 modified to include a declaration of the variable x (in the notation of [R], it is the union of U_1 and USING(VAR x)). <VAR – PAR>

(6.10) $\text{ALT}(c?x\ P, \underset{\sim}{G}) = \text{VAR } y : \text{ALT}(c?y\ \text{SEQ}(x := y, P), \underset{\sim}{G})$

provided $x \neq y$ and y is not free in P or $\underset{\sim}{G}$ <input renaming>

(6.11)‡ $\text{VAR } x : (\langle x \rangle + \underset{\sim}{y}) := (\langle e \rangle + \underset{\sim}{f}) = \text{VAR } x : (\underset{\sim}{y} := \underset{\sim}{f})$ <assignment elim>

(6.12) $\text{VAR } x : P = \text{VAR } x : \text{SEQ}(\text{VAR } z : (x := z), P)$ <initialisation>

(6.13) $\text{CHAN } c_1 \ldots c_n : P = P$ if none of $c_1 \ldots c_n$ appears free in P. <CHAN elim>

Laws of $\perp$

(7.1)* $\mathrm{ALT}(\mathrm{SKIP}\ \perp,\ \underline{G}) = \perp$ <ALT - SKIP zero>

(7.2)* $\mathrm{SEQ}(\perp,\ P) = \perp$ <SEQ left zero>

(7.3)* $\mathrm{SEQ}(\underline{x} := \underline{e},\ \perp) = \perp$ <SEQ right zero>

(7.4)* $\mathrm{PAR}(U_1 : \perp,\ U_2 : P) = \perp$ <PAR zero>

b) Some derived laws

(D1) $\mathrm{SEQ}(P,\ \mathrm{SEQ}(Q,R)) = \mathrm{SEQ}(\mathrm{SEQ}(P,Q),R)$ <SEQ binary assoc>

(D2) $\mathop{\mathrm{ALT}}_{i=1}^{n} g_i\ (\mathop{\mathrm{IF}}_{j=1}^{m} b_j\ P_{ij})) = \mathop{\mathrm{IF}}_{j=1}^{m} b_j\ (\mathop{\mathrm{ALT}}_{i=1}^{n} g_i\ P_{ij})$

providing $b_1 \vee \ldots \vee b_m \equiv$ true and no variable input in a g_i appears in a b_j. <ALT - IF distrib>

(D3) $\mathrm{SEQ}(\mathrm{SKIP},\ P) = \mathrm{SEQ}(P,\ \mathrm{SKIP}) = P$ <SEQ - SKIP unit>

(D4) $\mathrm{SEQ}(P,\ \mathop{\mathrm{IF}}_{i=1}^{n} b_i\ Q_i) = \mathop{\mathrm{IF}}_{i=1}^{n} b_i\ \mathrm{SEQ}(P,\ Q_i)$

if $b_1 \vee \ldots \vee b_n \equiv$ true and no variable in any b_i is altered by P. <SEQ - IF right distrib>

(D5) $\mathrm{PAR}(U_1{:}P,\ U_2{:}\mathrm{SKIP}) = \mathrm{PAR}(U_1{:}P) = P$

provided U_1 declares all global variables and channels used by P, and U_2 declares none of them <PAR - SKIP unit>

(D6) $\mathrm{SEQ}(\underline{x} := \underline{e},\ c!f) = \mathrm{SEQ}(c!f\left[{}^{\underline{e}}/_{\underline{x}}\right],\ \underline{x} := \underline{e})$ <assignment-output sym>

(D7) $\mathrm{SEQ}(\underline{x} := \underline{e},\ c?y) = \mathrm{SEQ}(c?y,\ \underline{x} := \underline{e})$

provided y is not free in $\underline{x} := \underline{e}$. <assignment-input sym>

(W1) WHILE b P = IF(b SEQ(P, WHILE b P), ¬b SKIP) <WHILE expansion>

(W2) WHILE b_1 (WHILE b_2 P) = WHILE $b_1 \vee b_2$ IF(b_2 P, <u>true</u> $\bot$)

<WHILE combination>

(W3) WHILE b P = IF(b WHILE <u>true</u> P, ¬b SKIP)

if no variable appearing in b is input or assigned to by P

<infinite loop>

(W4) WHILE <u>true</u> x:=e = $\bot$ <divergent loop>

(W5) WHILE b SEQ(P,Q)

= IF(b SEQ(P, WHILE b SEQ(Q,P), Q), ¬b SKIP)

if no variable appearing in b is input or assigned to in Q.

<WHILE reordering>

References

[AFR] Apt, K.R., Francez, N. and de Roever, W.P. A proof system for communicating sequential processes. ACM Trans. Program Lang. Syst. 2, 3 (July 1980) 359-385.

[B] Brookes, S.D. A model for communicating sequential processes. Oxford University D.Phil. thesis (1983). (Available as a Carnegie-Mellon University technical report.)

[D] Dijkstra, E.W. A discipline of programming. Prentice-Hall (1976).

[F] Fisher, A. Formal number theory and computability. Oxford Logic Guides 7, O.U.P. (1982).

[G] Guessarian, I. Algebraic semantics. Springer LNCS Vol. 99 (1981).

[H] Hoare, C.A.R. Communicating sequential processes. Prentice-Hall (1985).

[I] INMOS Ltd. The occam programming manual. Prentice-Hall (1984).

[laws] Hoare, C.A.R., He, Jifeng, Hayes, I.J., Morgan, C.C., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.A. and Roscoe, A.W. Laws of programming: a tutorial paper. Oxford University Computing Laboratory technical monograph PRG-45 (1985).

[R] Roscoe, A.W. Denotational semantics for occam. Version (a) in Proceedings of the July 1984 Seminar on Concurrency. Springer LNCS Vol. 197 (1985).

Version (b) to appear as a PRG monograph, Oxford University Computing Laboratory.

The semantics referred to in this paper is that of version (b). The only significant differences between these papers are in the treatment of uninitialised variables and in multiple outputs on the same channel: version (a) distinguishes between ALT(SKIP c!1, SKIP c!2) and ALT(c!1 SKIP, c!2 SKIP), but version (b) does not.

$VAR\,x \cdot ALT\, g_i : P_i$

$= VAR\,x : ALT\, g_i\, P_i^*$

where $P_i^* = P_i$ if x ... g_i

$= VAR\,x : P_i$ otherwise

$VAR\, x, y : ALT$
$c?x$
P_1
$c?y$
P_2
G

$= VAR\, x, y : ALT$
$c?x$
P_1
$c?x$
$P_2[x/y]$
G

x not free in P_2