PROCESSES: A MATHEMATICAL MODEL OF COMPUTING AGENTS

Robin MILNER

University of Edinburgh, Edinburgh, Scotland

Abstract

Most of the computing agents with which computing science is concerned, for example digital computers themselves, their memories and peripheral hardware devices, and–more abstractly–computer programs, exhibit a behaviour which is not just the computation of a mathematical function of their inputs, but rather a possibly infinite sequence of communications with their environment. Another way of viewing them is as transducers from sequences (of interrogation by, or responses from, the environment) to sequences (of responses to, or interrogations of, the environment).

A transducer is an intensional object; its extension or behaviour can be represented as a member of a domain P of *processes*. P is a complete lattice, satisfying the isomorphism

$$P \cong V \to (V \times P),$$

where V is a complete lattice, the domain of responses and interrogations, and '$\to$' forms the domain of continuous functions between two complete lattice domains. That P exists is a consequence of Dana Scott's inverse limit construction of a model D for the λ-calculus satisfying $D \cong D \to D$.

In as much as processes may be denoted by terms in an extended λ-calculus, we have a method of specifying the behaviour of computing agents, and also of formally demonstrating their properties. A crucial feature is the ability to define the operation of *binding* together two processes (which may represent two cooperating programs, or a program and a memory, or a computer and an input/output device) to yield another process representing the composite of two computing agents, with their mutual communications internalized.

The mathematics of processes promises to unify the behavioural study of computing agents, both hardware and software. As a contribution to the mathematical semantics of programming languages, this work is an extension of that of Strachey[17], Scott[16] and others at Oxford. Similar ideas have been put forward independently by H. Bekić[2] at the IBM Laboratory, Vienna.

1. Introduction

We may think of a computing agent as built, by certain constructions or composition operations, out of smaller components which are themselves computing agents. This very general notion includes as examples

(i) a memory register,

(ii) a memory, or store, as built from memory registers,

(iii) a central processor,

(iv) an input/output device, e.g. a line-printer,

(v) a digital computer, as built from a memory, a central processor and other components,

(vi) a computer network, as built from several digital computers, and more abstractly

(vii) a program instruction,

(viii) a program, as built from instructions; and we may even mix the levels of abstraction to obtain

(ix) the composition of a program and a line-printer.

We more or less understand many of the composition operations over these entities; the composition operation of example (ix) may make us feel uneasy, but we could argue that it is this composite object (not just the program) which the programmer presents as the solution to his problem – that is, its behaviour will meet his criteria for a correct solution.

This paper aims at solving the following problem: what (abstract) entities constitute the behaviours of computing agents in such a way that to discover the behaviour of a composite agent we may compose the behaviours of its components? A solution of this problem clearly has practical significance; the designer of a computing system should be able to think of his system as a composite of behaviours, in order that he may factor his design problem into smaller problems, each concerned with realizing some behaviour as the composite of subbehaviours and eventually by (more) concrete computing agents.

One area in which this problem has been almost satisfactorily solved is that of serial, deterministic programming languages. 'Serial' here means that the order of evaluation is fully determined. First, Strachey [17] and Landin [8] showed how to translate programs into the λ-calculus, and when Scott [10] constructed models for this calculus in terms of his continuous function spaces the effect was to ascribe to every program an abstract entity, a continuous function, as its meaning (see [16]). The technique of continuations, due independently to L. Morris and to C. Wadsworth and reported by Reynolds [13] made an important contribution in modelling features of programming languages which are not reflected in the λ-calculus, such as call-by-value, labels and jumps. At

least in simple languages the semantic function takes the form of a homomorphism from an algebra of programs to an algebra of continuous functions; this algebraic approach was explored first by Burstall and Landin[3], and in his thesis[12] L. Morris shows how it helps to structure a proof of compiler correctness. A machine-checked proof of compiler correctness, using functional semantics for both source and target languages, was carried out by Milner and Weyhrauch[11]; it also was algebraically structured.

There are, however, programming language features which have not been satisfactorily dealt with hitherto, and they are features whose analogue is frequently met in computing systems in general, both hardware and software. The following three are not necessarily all, but are some for which this paper attempts to find a solution:

(1) *Non-terminating programs with side-effects.* If the meaning of a program is taken to be a function from memories to memories (memory ≡ state-of-the-store) then a program like

while true **do** write (read( ))

will just mean the totally undefined function. This is clearly unsatisfactory; we need to capture the side-effect history of a program in its meaning.

(2) *Non-deterministic programs.* The meaning of a non-deterministic program is not obviously a (single-valued) function. Some authors have therefore introduced binary relations as meanings; see for example [4, 5, 7]. There are some difficulties yet to be resolved in this approach, however. If S is the domain of memories, the meaning MNG(e) of a program e will be a subset of $S \times S$. Then we can have MNG(e) = MNG(e') even if e always terminates while e' has an extra option which allows it to loop. To capture this difference we may make S into a partial order by adjoining an 'undefined' minimum element $\perp$, and then for some s, $\langle s, \perp \rangle \in$ MNG(e') while $\langle s, \perp \rangle \notin$ MNG(e). Now we have an embarassing richness of possible relationships between meanings, involving set-theoretic inclusion as well as the ordering relation on S. This reflects a real complexity of non-deterministic programs (a point well brought out by Manna[9]), and if there is a satisfactory generalization to relations of Scott's notion of continuous function it has yet to be found.

(3) *Parallel programs.* Consider two program statements

$$e = \text{'}\textbf{begin}\ x := 0;\quad x := x + 1\ \textbf{end}\text{'}$$

$$e' = \text{'}x := 1\text{'}$$

Both e and e' mean the same as functions (or relations) over memories. But executing e and e' in pseudo-parallel (allowing arbitrary interleaving of execution sequences) can result in a memory in which x has the value 1, or the value 2, while executing e' in pseudo-parallel with another copy of itself can yield only the value 1 for x. So in general the meaning of the program statement 'e **par** e'' is not fully determined by the meanings of e and e' as subsets of $S \times S$. The memory-access history of programs must be included in their meanings if we are to be able to compound the meaning of a parallel program from the meanings of its components.

Now in the case of (1) and (3) above we saw that the meaning of a program should express its history of access to resources which are not private to it. This is also true, perhaps more clearly, for the meanings or behaviours of hardware devices. We may go further and say that the overt behaviour constitutes the *whole* meaning of any computing agent. We shall arrive at a suitable mathematical domain of behaviours by first considering any computing agent as a transducer, whose input sequence consists of enquiries by, or responses from, its environment, and whose output sequence consists of enquiries of, or responses to, its environment.

The parallel composition of two computing agents will be represented by a behaviour consisting of all possible ways of merging the behaviours of the components. This entails an element of non-determinism (see also [1], where parallelism is explicated in terms of non-determinism). We shall handle the latter in general by introducing fictitious agents called oracles, whose function is to provide a sequence of truth values used to resolve the arbitrary choices occurring in a behaviour, thus allowing us to represent behaviours as single-valued functions in a certain domain. Given a satisfactory generalization of Scott's continuous function spaces to relations, we could dispense with oracles, while retaining the remainder of the model of behaviour to be presented.

2. Transducers and processes

We consider first how a transducer can be used to model a computing agent capable of communicating with any number of others. We presup-

pose two sets L and V, respectively a set of addresses used to label the communication lines of a transducer and a set of values used in communication.

A (Mealy-type) transducer is a quadruple $\langle S, s_0, f, g\rangle$ where S is a set of states, $s_0 \in S$ is a starting state,

$$f : S \longrightarrow (V \longrightarrow (L \times V))$$

is the output function, and

$$g : S \longrightarrow (V \longrightarrow S)$$

is the state transition function. Thus, in a given state with a given input value, the output function determines both the output communication line and the value to be transmitted on that line. Although we have allowed many output lines we have apparently allowed only one input line. However this restriction is only apparent; if we wish we may take the identity of the input line to be encoded in the input value. I have chosen to treat input and output asymmetrically in that the identity of the output line is made explicit, but this seems to be only a matter of expediency.

In the usual way we can extend f and g to

$$f^* : S \longrightarrow (V^* \longrightarrow (L \times V)^*),$$
$$g^* : S \longrightarrow (V^* \longrightarrow S),$$

and we can extend f to

$$f^\infty : S \longrightarrow (V^\infty \longrightarrow (L \times V)^\infty)$$

where for any set A, A^* and A^∞ are respectively the sets of finite and of infinite sequences over A. We are not in general concerned with arbitrary input sequences in V^∞, but rather sequences $\bar{u} = \langle \bar{u}_i \mid i \geq 0\rangle$ such that u_{i+1} is dependent on $\langle \alpha_i, v_i\rangle \in L \times V$, where $f^\infty(\bar{u}) = \langle\langle \alpha_i, v_i\rangle \mid i \geq 0\rangle$; that is, the output $\langle \alpha_i, v_i\rangle$ is often taken as sending v_i as input to another transducer addressed by α_i, whose eventual response will be taken as u_{i+1}.

Now it is easy enough to derive from two transducers T_1, T_2 a composite transducer T which results from making T_1 and T_2 intercommunicate in the manner suggested in the last paragraph. But the behaviour–the infinite output function $f^\infty(s_0)$–of T is easily seen to depend only on the behaviours of T_1 and T_2. We may therefore concentrate on first representing behaviours satisfactorily, and then defining composition operations on behaviours.

We might take all functions $V^\infty \to (L \times V)^\infty$ as the domain of

behaviours; however, we wish to restrict the domain to those functions f for which the i^{th} component of $f(\bar{u})$ depends, not simply on only some finite initial segment of $\bar{u}$ (these would be the continuous functions) but more specifically only on the i^{th} initial segment of $\bar{u}$. The following therefore seems more satisfactory.

First, we take the domains L, V and S to be complete lattices – if necessary by adjoining to each a top ('overdefined') and bottom ('underdefined') element, respectively $\mathsf{T}_L, \perp_L, \mathsf{T}_V, \perp_V$ etc. Then the work of Scott [15] ensures the existence of a domain P of *processes* satisfying the order isomorphism

$$P \cong V \longrightarrow (L \times V \times P),$$

that is, a process is a function which given a value u produces a pair $\langle \alpha, v \rangle \in L \times V$ and a new process. Given a transducer $\langle S, s_0, f, g \rangle$, we can then represent its behaviour by the process $\mathrm{BEH}(s_0)$, where $\mathrm{BEH} : S \to P$ is defined recursively by

$$\mathrm{BEH}s = \lambda u \cdot \langle fsu, \mathrm{BEH}(gsu) \rangle.$$

[Note. fsu is a pair $\in L \times V$. Except between capital letters, we use juxtaposition to represent function application and assume it to associate to the left.] More precisely, using the fixed point operator Y,

$$\mathrm{BEH} = Y\lambda F.(\lambda s.\lambda u.\langle fsu, F(gsu) \rangle)$$

It can be shown that for two transducers $\langle S, s_0, f, g \rangle$ and $\langle S', s_0', f', g' \rangle$

$$f^{\infty}(s_0) = f'^{\infty}(s_0') \Leftrightarrow \mathrm{BEH}(s_0) = \mathrm{BEH}'(s_0')$$

which justifies our representing behaviours as processes. Moreover for *any* process p there exists a transducer whose behaviour is p.

We introduce a little notation and terminology associated with processes. Let $C = (L \times V) \times V$ be the domain of *communications*.

We define the relation $t \overset{\gamma}{\Longmapsto} t'$, where $t, t' \in L \times V \times P$ and $\gamma \in C^*$, as follows:

(i) If $\epsilon \in C^*$ is the null sequence, then $t \overset{\epsilon}{\Longmapsto} t$.

(ii) If $t = \langle \alpha, v, p \rangle$ and $pu \overset{\gamma}{\Longmapsto} t'$, then $t \overset{\langle \langle \alpha, v \rangle, u \rangle \gamma}{\Longmapsto} t'$.

If $pu \overset{\gamma}{\Longmapsto} \langle \alpha, v, p' \rangle$ we call γ a *communication sequence* of p (under u), and p' a *renewal* of p (this term was used with a slightly different meaning

in my previous paper [10]). Finally, if $\gamma' \in C^{\infty}$ and every initial segment of γ' is a communication sequence of p (under u), we call γ' an *infinite communication sequence* of p (under u).

Communication sequences are related to the extended output function of a transducer as follows: if for transducer $T = \langle S, s_0, f, g \rangle$ we have

$$f^{\infty}(s_0)(\langle u_0, u_1, \ldots \rangle) = (\langle \alpha_0, v_0 \rangle, \langle \alpha, v_1 \rangle, \ldots).$$

then BEH(s_0), the behaviour of T, will have under u_0 the communication sequence

$$\langle \langle \alpha_0, v_0 \rangle, u_1 \rangle, \langle \langle (\alpha_1, v_1 \rangle, u_2 \rangle, \ldots .$$

We can now show how a simple program statement e, '$x := y$' for example, can be given meaning as a process. We suppose that V contains a distinguished element '!', and L a distinguished element 'ι' (their purpose will become apparent below). Suppose also we are executing in an environment which associated the identifiers x, y respectively with α, $\beta \in L$. The process p modelling e will involve communications with addresses α and β; we will later show how to *bind* resources (memory registers in this case) to α and β in p, but p embodies the meaning of e before this binding. We define p by giving its communication sequences:

$$pu \overset{\langle\langle \beta, ! \rangle, v \rangle\langle\langle \alpha, v \rangle, w \rangle}{\Longrightarrow} \langle \iota, w, \perp_p \rangle,$$

where u, v and w are arbitrary members of V. Alternatively,

$$p = \lambda u.\langle \beta, !, \lambda v.\langle \alpha, v, \lambda w.\langle \iota, w, \perp_p \rangle \rangle \rangle$$

Now if we assume that a memory register, given input '!', will in some sense respond with its stored value v ($\neq$ '!'), and given any other input v will store v and respond with v, then the effect of binding memory registers as resources to α and β in p will be (as a result of our later definition of 'resources' and 'binding') that the desired assignment operation has occurred and moreover that the assigned value is sent to the result address ι. This latter effect is what we would want for a programming language in which an assignment is an expression (with side-effects) whose value is the value assigned.

To see how this works in greater detail, the reader may wish at this point to refer forward to the section called 'Resources and Binding', which may be read independently of the intervening sections. He may

then check that the result of binding to β in p a register, initially containing 5 say, is the process

$$\lambda u.\langle \alpha, 5, \lambda w.\langle \iota, w, \perp_p \rangle \rangle$$

and that further binding a register to α yields the process

$$\lambda u.\langle \iota, 5, \perp_p \rangle.$$

The side-effect of the assignment to the register bound to α is not yet clear. But in the more realistic situation in which the scope of the program variables x and y is larger than just the statement e–for example, the scope may be '$e\,;e'$', where e' is some statement–we will bind our registers to the composite of the meanings of e and e' (serial composition of processes is defined in the next section) and then e' will 'feel' the side-effect of e upon x.

A notion similar to process, as defined here, has been studied independently by H. Bekić[2]. Processes are a generalization of the notion of 'stream' introduced by Landin[8].

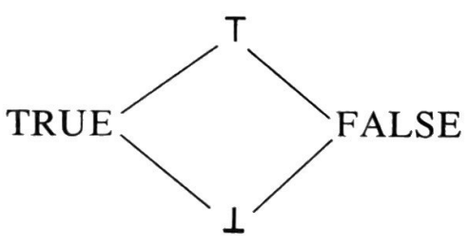
3. Combinations of processes, and oracles

We require that the domain V contains the truth-values domain T, and we shall also need it to contain $W = V \times V$, the domain of pairs of values, and the domain L. So

$$V = L + T + W + U,$$

where we don't specify U further.

T is the domain



We need notation for the injection and projection functions between a disjoint union domain and its disjunct domains, and also discriminator predicates which determine in which disjunct domain an element lies. We denote by '**in**V' all the injection functions from $L, T, \ldots$ into V, and we postfix these functions. The corresponding projection functions (also

postfixed) we name '$| L$', '$| T$', etc. For example

$$v \mid L = \begin{cases} \mathsf{T}_L & \text{if } v = \mathsf{T}_V, \\ \alpha & \text{if } v = \alpha \,\mathbf{in}\, V \text{ for some } \alpha \in L, \\ \bot_L & \text{otherwise.} \end{cases}$$

The corresponding discriminator functions (also postfixed) we name '$:L$', '$:T$,' etc. For example

$$v : L = \begin{cases} \mathsf{T}_T & \text{if } v = \mathsf{T}_V, \\ \bot_T & \text{if } v = \bot_V, \\ \text{TRUE} & \text{if } v = \alpha \,\mathbf{in}\, V \text{ for some } \alpha \in L, \\ \text{FALSE} & \text{otherwise.} \end{cases}$$

For each domain A we define the conditional operation '$\supset$' $\in T \to A \to A \to A$ by $\supset xyz = y, z, \mathsf{T}_A, \bot_A$ according as $x =$ TRUE, FALSE, T_T, $\bot_T$, and we write as usual $x \supset y, z$ for $\supset xyz$.

Finally, we use the notation $\langle \text{-}, \text{-} \rangle$, $\langle \text{-}, \text{-}, \text{-} \rangle$, $(\,)_1$, $(\,)_2$, $(\,)_3$ for pairing, tripling and selector functions, omitting parentheses in the last three when the argument is an atomic term.

We now introduce three binary operations on processes, giving us a simple algebra of processes. We use the infixed symbols '$*$', '$?$' and '$\|$'.

(1) *Serial composition*

The operation '$*$' is defined recursively as follows

$$p * p' = \lambda v.((pv)_1 = \iota) \supset p'(pv)_2, \quad \langle (pv)_1, (pv)_2, (pv)_3 * p' \rangle.$$

In other words, if ι does not occur in γ, then

$$pv \overset{\gamma}{\Longmapsto} \langle \alpha, w, p'' \rangle \text{ implies } (p * p')v \overset{\gamma}{\Longmapsto} \begin{cases} p'w & \text{if } \alpha = \iota, \\ \langle \alpha, w, p'' * p' \rangle & \text{otherwise.} \end{cases}$$

Thus informally, p is executed until it produces a result w, which is then fed to p'. It's easy to show that '$*$' is associative, using properties of the domain P.

(2) *Choice of composition*

$$p \ ? \ p' = \lambda v.\langle \omega, !, (\lambda u.u \mid T \supset pv, p'v) \rangle$$

We imagine that an oracle is a resource (in the sense to be later defined) which responds always with TRUE **in** V or FALSE **in** V, and that $\omega \in L$ is

a distinguished address used for consulting an oracle. Here the response from the oracle is to be used to decide whether to do p or p'.

(3) *Parallel composition*

We want for $p \parallel p'$ that, given a pair $\langle w_1, w_2 \rangle$, if either p or p' reaches a result–i.e., $(pw_1)_1 = \iota$ or $(p'w_2)_1 = \iota$–then the other is completed and the two results paired, otherwise the oracle is consulted to determine which of their communications to perform. So let $w = v \mid W$ and $s = pw_1$, $t = p'w_2$; then the operation '$\parallel$' is defined recursively as follows:

$$\begin{aligned}(p \parallel p')v = {} & (s_1 = \iota) \supset (p' * \lambda u.\langle \iota, \langle s_2, u\rangle \textbf{in } V, \bot\rangle)w_2, \\ & (s_2 = \iota) \supset (p * \lambda u.\langle \iota, \langle u, t_2\rangle \textbf{in } V, \bot\rangle)w_1, \\ & \langle \omega, !, \lambda x.x \mid T \supset \langle s_1, s_2, \lambda u.(s_3 \parallel p')(\langle u, w_2\rangle \textbf{in } V)\rangle, \\ & \qquad \langle t_1, t_2, \lambda u.(p \parallel t_3)(\langle w_1, u\rangle \textbf{in } V)\rangle\rangle.\end{aligned}$$

Another form of parallel composition, which 'terminates' when one or other (not both necessarily) of the composed processes 'terminates', is easily defined in a similar manner.

4. Semantics of programs, and criteria for adequacy

Consider a language B generated from a set $A = \{a_1, a_2, ...\}$ of atomic statements by the binary infixed operations ';' '**or**' and '**par**'. (B, ';', '**or**', '**par**') is an algebra; we may take it as a word algebra, or impose on the operations just one relation, the associativity of ';'. Examples of programs in B are

$$a_1 \,;\, a_2 \,;\, a_2$$

$$(a_1 \textbf{ or } (a_2 \,;\, a_3)) \textbf{ par } (a_2 \,;\, a_4)$$

Now we give the three syntactic operations their intended meanings as respectively serial, choice and parallel composition by uniquely extending a given semantic function $\mathcal{M} : A \to P$ to a homomorphism from (B, ';', '**or**', '**par**') into the algebra (P, '$*$', '?', '$\parallel$'). The extension is of course possible due to the associativity of '$*$' over processes.

If B is enriched by the following operations: conditionals (**if** .. **then** .. **else** ..), iterations (**while** .. **do** ..), declarations (including procedures) and block structure, it is still possible to find corresponding operations over P which are *intuitively* correct, and to extend $\mathcal{M}$ to a homomorphism as before. For

details of this, see [10]. In the present paper I want to highlight the operation of binding resources to processes, but first to divert to the general questions: what is an *adequate* semantic function? I restrict the discussion to programming languages, although it is relevant to the more general question: what is an adequate abstraction of the operation by which we compose any computing agents?

If we accept that any abstract semantics should give a way of composing the meanings of parts into the meaning of the whole, this is just to say that the semantic function of a programming language which is an algebra should be a homomorphism, or equivalently that it should induce a congruence relation in that algebra. But programming languages are usually designed in the first place *operationally*; it is only by specifying some evaluation mechanism (possibly by an abstract machine) that we can be sure that a language can be practically implemented. So it is not just a lack of mathematical sophistication that has caused all existing practical languages to be defined by evaluation rules.

As an example, an operational semantics for the language B defined above may give an evaluation function $\mathscr{E} : B \to (S \to 2^S)$, where S is the domain of memories. In this way, evaluation determines an equivalence relation on programs; $e \sim e'$ iff $\mathscr{E}(e) = \mathscr{E}(e')$. But this relation may not be a congruence; the trivial example given in the introduction showed that $e \sim e'$ does not imply $(e \textbf{ par } e') \sim (e' \textbf{ par } e')$.

Let us say that a semantic function $\mathscr{M}$ for B is *adequate w.r.t.* $\mathscr{E}$ iff $\mathscr{M}$ induces a congruence $\approx$ on B which respects $\sim$. That is, we require for all $e, e' \in B$

$$e \approx e' \Rightarrow e \sim e'.$$

This permits too much however; if we take $\mathscr{M}$ to be the identity function over B it induces the identity congruence relation (two programs mean the same iff they are identical!) which clearly respects any equivalence relation.

But among the congruences respecting $\sim$ there is a unique maximum congruence $\approx_{\mathscr{E}}$ defined by

$$e \approx_{\mathscr{E}} e' \Leftrightarrow \forall \text{ contexts } \mathscr{C}.\mathscr{C}(e) \sim \mathscr{C}(e'),$$

where a context is any derived unary operation in the algebra B. That is, $\approx_{\mathscr{E}}$ respects $\sim$, and any congruence $\approx$ respecting $\sim$ respects $\approx_{\mathscr{E}}$ also. So

we say $\mathcal{M}$, a semantic function for B, is *fully abstract* w.r.t. $\mathscr{E}$ iff for all $e, e' \in B$

$$\mathcal{M}(e) = \mathcal{M}(e') \Leftrightarrow e \approx_{\mathscr{E}} e'$$

and this defines $\mathcal{M}$ uniquely in the sense that if $\mathcal{M}, \mathcal{M}'$ are fully abstract w.r.t. $\mathscr{E}$, then the algebras $\mathcal{M}(B)$ and $\mathcal{M}'(B)$ are isomorphic.

This is not a constructive definition of $\mathcal{M}$; but we should hope to prove for any constructive $\mathcal{M}$ (e.g. λ-definable in terms of processes) in the first place that it is adequate, and ideally that it is fully abstract, w.r.t. a given $\mathscr{E}$.

It is unfortunate that the semantics we have given for B, although probably adequate, is not fully abstract for any natural $\mathscr{E}$. For such an $\mathscr{E}$ will entail $(e \textbf{ or } e') \sim (e' \textbf{ or } e)$; on the other hand it is easy to see that $\mathcal{M}(e \textbf{ or } e') \neq \mathcal{M}(e' \textbf{ or } e)$ for the trivial reason that the two processes use the oracle differently. In this particular case therefore the question remains open of exhibiting the fully abstract semantic function for B w.r.t. a suitable $\mathscr{E}$, and in general I believe the criteria of adequacy and full abstraction should be used to assess semantic functions.

5. Resources and binding

The processes in $\mathcal{M}(B)$, the set of meanings of our language B, will make communication both to ω (for oracles) and to other addresses, depending on the set $\mathcal{M}(A)$ of meanings of atomic statements. We want to show how to bind resources to these addresses.

One method of modelling resources is as processes. As an example, suppose a memory register storing the value v is modelled by the process REG(v). With the convention that an input '!' to such a process means 'FETCH' and any input v ($\neq$!) means 'STORE v', we can define REG $\in V \to P$ recursively by

$$\text{REG } v = \lambda u.(u = !) \supset \langle \iota, v, \text{REG } v\rangle, \langle \iota, u, \text{REG } u\rangle.$$

Note that the renewal is a new (abstract) memory register. In [10], I showed how to define an operation

$$\text{BIND} \in L \to P \to P \to P$$

so that for example BIND μp(REG v) (which we here abbreviate to $p_{\mu v}$) has communication sequences given as follows: if μ does not occur in γ,

then

$$pu \overset{\gamma}{\Longrightarrow} \langle \alpha, w, p' \rangle \text{ implies } p_{\mu v}u \overset{\gamma}{\Longrightarrow} \begin{cases} p'_{\mu v}v & \text{if } \alpha = \mu,\ w = !, \\ p'_{\mu w}w & \text{if } \alpha = \mu,\ w \neq !, \\ \langle \alpha, w, p'_{\mu v} \rangle & \text{otherwise.} \end{cases}$$

Thus BIND $\alpha pp'$ – the result of binding p' to α in p – is in general a process in which all communications to α in p are served by the resource p', and so 'internalised'.

With this definition, BIND $\alpha p \in P \to P$ gives the extension of p as a resource user (through α). There is an analogy with λ-abstraction. The process which results from giving a resource to a user is gained by applying the latter to the former. However, there is here an asymmetry between resources and resource users which we may wish to eliminate. For example: which is the resource and which the user, of two coroutines or of two computers intercommunicating? The best answer seems to be: whichever has control at a given instant. So if resource and user are (abstractly) the same kind of object, and putting them together by application yields a process, we are led to consider modelling both by the domain Q satisfying the isomorphism

$$Q \cong Q \to P$$

which again may be constructed by Scott's method. We may now redefine BIND : $L \to P \to Q$ recursively by

$$\text{BIND } \alpha p = \lambda q.\lambda v.((pv)_1 = \alpha) \supset q(\text{BIND } \alpha(pv)_3)(pv)_2, \langle (pv)_1, (pv)_2, \text{BIND } \alpha(pv)_3 q \rangle.$$

We may call BIND αp a 'resource abstraction' of p. Applied to an argument resource q, it becomes a process which invokes q – by giving it as resource a resource abstraction of a renewal of p – just when p would pass control along the communication line labelled α. This is a generalization of the notion of continuations mentioned in the introduction, in which for example calling a procedure may be modelled by applying it to the continuation of the caller.

As an example of an explicit resource definition, consider the memory register again; we redefine REG$(v) \in$ Q recursively by

$$(\text{REG } v)q = \lambda u.(u = !) \supset q(\text{REG } v)v, q(\text{REG } u)u$$

Another example is oracles; we call $\Omega \in Q$ an *oracle* if

$$\Omega = \lambda q.\lambda v.q\,\Omega' u,$$

where u is either TRUE **in** V or FALSE **in** V and Ω' is again an oracle, independent of q, v. This defines oracles as a subset of Q.

Now suppose the atomic statements A of our language B are assigned as meanings processes which only use a single memory address μ, as well as the addresses ι and ω, and all of whose infinite communication sequences contain ι. Then any $p \in \mathcal{M}(B)$ will have the same property; in fact, it may be shown that if $p = \mathcal{M}(e)$, $e \in B$, and

$$p' = \text{BIND } \omega(\text{BIND } \mu p(\text{REG } 0))\Omega,$$

where Ω is any oracle, then p' represents a function $\in V \to V$, in the sense that for any $v \in V$ we have

$$p'v = \langle \iota, u, p'' \rangle \quad \text{for some } u \in V.$$

So $\lambda v.(p'v)_2$ is the function computed by e (under a particular choice discipline imposed by Ω). We would hope to show that the union of all these functions, as Ω ranges over all oracles, is the relation $\subseteq V \times V$ assigned to e in some operational semantics for B; this would amount to a proof that $\mathcal{M}$ is adequate w.r.t. the operational semantics.

It was shown in [10] how a local program variable can be represented, as we would expect by binding an appropriate resource to the process which is the meaning of that part of the program constituting the scope of the variable.

Let us turn now to the question of intercommunicating processes in general. For two processes p, p' we may form the process

$$p'' = (\text{BIND } \alpha p)(\text{BIND } \alpha' p').$$

Informally, p is given control first, and control will pass back and forth between p and p' via the addresses α and α'. This composition is clearly applicable in the case of coroutines in programming.

More generally, suppose we have $n + 1$ processes $p_0, \ldots, p_n$, addressing each other by $\alpha_0, \ldots, \alpha_n$. The composite process does not seem to be representable by our present definition of BIND, but we can generalize it in a natural way, as follows. First, we introduce the domains Q_n, each defined by the isomorphism

$$Q_n \cong \overbrace{Q_n \to Q_N \to \cdots \to Q_n}^{n \text{ times}} \to P = Q_n^n \to P.$$

To define $\mathrm{BIND}_n \in L \to P \to Q_n$ we introduce for convenience the following notation: if $\bar{x} = x_0 x_1 \ldots x_n$ is any sequence of $n+1$ terms, then $\bar{x}^{(i)} = x_i x_{i+1} \ldots x_n x_0 \ldots x_{i-1}$, the i^{th} cyclic permutation of $\bar{x}$ $(0 \le i \le n)$. Now define BIND_n recursively by

$$(\mathrm{BIND}_n\, \bar{\alpha} p q_1 \ldots q_n)v = \begin{cases} \bar{r}^{(i)}(pv)_2 & \text{if } (pv)_1 = \alpha_i, \quad 0 \le i \le n, \\ \langle (pv)_1, (pv_2), \bar{r} \rangle & \text{otherwise,} \end{cases}$$

where

$$\bar{r} = (\mathrm{BIND}_n\, \bar{\alpha}\, (pv)_3) q_1\, q_2 \ldots q_n.$$

(In this definition we have actually allowed p to communicate with *itself* through α_0, so the case $n = 1$ does not quite yield our original BIND).

We can call $\mathrm{BIND}_n\, \bar{\alpha} p$ and 'n-ary resource abstraction' of p. We can express the composite of the processes $p_0, \ldots, p_n$, in which p_0 is given control first and thereafter control is passed to p_i via address α_i, simply by the application of $n+1$ n-ary resource abstractions:

$$(\mathrm{BIND}_n\, \bar{\alpha}^{(0)} p_0)(\mathrm{BIND}_n\, \bar{\alpha}^{(1)} p_1) \ldots (\mathrm{BIND}_n\, \bar{\alpha}^{(n)} p_n).$$

So $\mathrm{BIND}_n\, \bar{\alpha} p$ gives the extension of p as a member of an $n+1$-ary composition of processes.

6. Discussion

Much of this paper has taken the form of a description of concepts which may be useful in modelling the behaviour of intercommunicating agents, together with some examples. The reader may well feel dissatisfied on two counts; first, that we have given no example of a proof that a system (e.g. a program) whose meaning is expressed as a process, performs a certain task correctly, and second that we are far from having a theory of processes–I have done not much more than say what they are.

Nevertheless I have tried to show that processes are worth the effort of constructing a theory, in that they model the behaviour of computing agents in a natural way. Moreover it is becoming clear that informal (but rigorous) proofs about communicating systems, analogous to proofs about serial programs based on the assertion technique introduced by Floyd[6] and others, are considerably harder to express (and to accept)

than the latter. A theory, which among other things isolates interesting subclasses of processes, will be essential for such proofs. (For example, a property of processes which shows signs of being useful is *insensitivity*: a process is insensitive if it is a constant function of its input, and if all its renewals are insensitive). Until a theory emerges, it is too early to consider a formal deductive system in which proofs may be executed (and of course checked by a machine), though it may well be that Scott's calculus LAMBDA[15] is a good starting point.

It is hoped that processes will give a firm basis for understanding and comparing the wide variety of existing and proposed features for communication and parallelism, including lockouts, protection, interrupts, semaphores, queues, priorities, etc. The extent to which they succeed in this is a measure of their importance.

Acknowledgements

I would like to thank Rod Burstall, Jean-Marie Cadiou, Gilles Kahn, Jim Morris, Jim Thatcher and Richard Weyhrauch for helpful discussions.

References

[1] E. Ashcroft and Z. Manna, Formalization of properties of parallel programs, Artificial Intelligence Memo AIM-110, Stanford University, Stanford, Calif. (1970).

[2] H. Bekić, Towards a mathematical theory of processes, Technical Report TR25.125, IBM Laboratory, Vienna (December 1971).

[3] R. M. Burstall and P. Landin, Programs and their proofs; and algebraic approach, in: B. Meltzer and D. Michie, eds., *Machine Intelligence*, Vol. 4 (Edinburgh University Press, Edinburgh, 1969) pp. 17–43.

[4] J. W. de Bakker, Recursive procedures, Mathematical Centre Tracts 24, Mathematical Centre, Amsterdam (1971).

[5] J. W. de Bakker and W. P. de Roever, A calculus for recursive program schemes, in: M. Nivat, ed., *Automata, languages and programming* (North-Holland, Amsterdam, 1973) pp. 167–196.

[6] R. W. Floyd, Assigning meanings to programs, in: J. T. Schwartz, ed., *Proceedings of the Symposium in Applied Mathematics*, Vol. 19 (Am. Math. Soc., Providence, R.I., 1967) pp. 19–32.

[7] P. Hitchcock and D. M. R. Park, Induction rules and termination proofs, in: M. Nivat, ed., *Automata, Languages and Programming*, (North-Holland, Amsterdam, 1973) pp. 225–251.

[8] P. J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, *Communications of the Association for Computing Machinery* **8** (1965) 89–101; 158–165.

[9] Z. Manna, The correctness of non-deterministic programs, *Artificial Intelligence* **1** (1970) 1–26.

[10] R. Milner, An approach to the semantics of parallel programs, Proc. Convegno di Informatica Teorica, Pisa, March 1973.

[11] R. Milner and R. Weyhrauch, Compiler correctness in a mechanized logic, in: D. Michie, ed., *Machine intelligence*, Vol. 7 (Edinburgh University Press, Edinburgh, 1972).
[12] L. Morris, Correctness of translation of programming languages–an algebraic approach, Artificial Intelligence Memo. AIM-174, Stanford University, Stanford, Calif. (1972).
[13] J. C. Reynolds, Definitional interpreters for higher order programming languages, *Proceedings of the ACM National Conference*, Boston 1972.
[14] D. Scott, Models for various type-free calculi, in: P. Suppes, ed., *Logic, Methodology and Philosophy of Science* IV (North-Holland, Amsterdam, 1973) pp. 157–8.
[15] D. Scott, Data types as lattices, unpublished Lecture Note (1972).
[16] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, *Proceedings of the Symposium on Computers and Automata*, Microwave Research Institute Symposia Series, Vol. 21, Polytechnic Institute of Brooklyn (1971).
[17] C. Strachey, Towards a formal semantics, in: T. B. Steel, Jr., ed., *Formal Language Description Languages for Computer-Programming* (North-Holland, Amsterdam, 1966).