Communication is an abstraction

Gérard Boudol

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 636

COMMUNICATION IS AN ABSTRACTION

Gérard BOUDOL

Mars 1987

Communication is an Abstraction

Gérard Boudol
INRIA Sophia-Antipolis
06560-VALBONNE FRANCE

LA COMMUNICATION EST UNE ABSTRACTION

Abstract.

We introduce a formalism that allows to hierarchically define communication structures. The core language consists of a few imperative constructs operating on shared variables. The semantics describes two distinct behaviours: a program performs some actions, which are elementary instructions, and operates on data by means of its terminated sequences of actions. By abstracting from this operation we get the notion of atom. At the abstract level an atom operates as an indivisible action, while it is implemented as a whole program of a lower level. We show that the concept of atom is well-suited to deal with both low-level and high-level communication and synchronization primitives.

Résumé.

Nous présentons un langage, et sa sémantique, dans lequel on peut introduire des mécanismes de communication et de synchronisation à différents niveaux. Le noyau du langage est constitué de quelques primitives de programmation impérative, qui manipulent des variables partagées; il n'y a pas d'autre moyen de communication primitif dans le langage proposé. Dans la sémantique nous décrivons deux modes de fonctionnement des programmes: d'une part ceux-ci exécutent des actions élémentaires, d'autre part ils opèrent, par leurs séquences d'actions terminées, sur des données. Un mécanisme permet d'abstraire l'opération d'un programme en celle, indivisible à son niveau, d'un atome. Nous montrons que cette notion d'atome permet de rendre compte de primitives de communication de différents niveaux.

1. Introduction.

From semaphores and critical regions to monitors, some programming concepts have been designed to insure a mutually exclusive use of resources – data or devices. Various message passing primitives have been used to implement communication in distributed systems: communication may be asynchronous (bufferized) or synchronous, and it may be one-to-one (e.g. rendez-vous), or many-to-one (client-server), or one-to-many (broadcast), and so on. Usually programming languages offer one of these primitives, but no way to introduce various synchronization mechanisms or communication structures – two exceptions being monitors, with a fixed synchronization discipline, and tasks. The purpose of this paper is to set up a semantical framework that embodies the idea of abstraction, from the low level shared variables to higher level communication schemes.

This research is partially supported under PRC C^3 (CNRS).

Obviously the main goal of language design is to conceive high-level constructs providing facilities for programming and maintaining large systems – whereas language development should provide efficient implementation. From a theoretical point of view this means that one has to search for powerful constructive concepts with a mathematically simple semantics – a good example is type theory (*cf.* [7] for instance). Regarding communication there are at least two mathematically well-founded ideas: deterministic stream processing (Kahn-MacQueen networks) and rendez-vous (CCS communication); we are mainly interested in the last one. In Milner's CCS the co-occurrence of sending and receiving a value v on the same port α, that is the co-occurrence of $(\alpha!v)$ and $(\alpha?v)$ gives rise to the so-called τ event. Later on Milner gave the suitable formalization: in SCCS [11] the co-occurrence of two actions is their (commutative) product, and sending and receiving are inverse actions; this provides a powerful theory (*cf.* [4]). However from an operational point of view the communication law, which may now be written $\alpha_v \cdot {\alpha_v}^{-1} = 1$, is some kind of miracle. Formally we could say that nothing allows to *prove* that such a communication occurs; more intuitively the question is: how to implement such a synchronization?

In fact there is an answer (*cf.* [8]); let us deal with pure synchronization, without value passing: $\alpha \cdot \alpha^{-1} = 1$. Then to make a CCS port one takes two boolean semaphores s and s' (initially true) and defines

$$\alpha = P(s)\,;V(s')$$
$$\alpha^{-1} = P(s')\,;V(s)$$

where P and V are Dijkstra's operations

$$P(s) = (\text{when } s \text{ do } s{:=}\mathit{ff})$$
$$V(s) = (\text{when } \neg s \text{ do } s{:=}\mathit{tt})$$

If we assume that P and V are *uninterruptible* and *mutually exclusive* operations, and that α and α^{-1} are the only way to use s and s', then it is easily seen that one cannot complete, say, an α without engaging "simultaneously" an α^{-1}. Moreover we must assume that performing, say, α^{-1} means to complete it; we thence postulate that it is an *atomic* (or more accurately "recoverable") action, abstracted from its code, which could very well be a complex program. Then the so-called "mutual inclusion" of α and α^{-1} results from an indissoluble interleaving of their codes, as in the sequence $P(s)\,;P(s')\,;V(s')\,;V(s)$ for instance.

This paper introduces a formalism to deal with such a way of looking at communication and synchronization. The starting point is an imperative language – for there is no simple functional semantics of concurrent and communicating processes. The primitive constructs are the following:

- simultaneous multiple assignments: $(x_1, \ldots, x_n){:=}(e_1, \ldots, e_n)$;
- boolean guards: (when t do p);
- sequential and parallel composition: $p\,;q$ and $(p \,\|\, q)$.

We could add boolean conditionals, or more generally case statements. There is no explicit communication primitive in this core language: at the deepest level *shared variables* are the only communication means. We shall see later the other constructs of the proposed formalism; let us say at this point a few words about semantics.

A program behaves in two different ways: first, given some environment collecting definitions (of data types, recursive processes, and so on), a program may perform some elementary *actions*. This is formalized as the "execution" transition relation

$$\mathit{environment} \quad \vdash \quad \mathit{program} \xrightarrow[\text{exec}]{\mathit{action}} \mathit{program}'$$

This transition relation is a rather "symbolic" one: it only says what is a possible next elementary thing to do. For instance if a and b are actions then the program $a\,;b$ first perfoms a, then b, and terminates. Such a notion of (labelled) transitions is usually taken as a suitable framework to describe the operational semantics of concurrent and communicating systems – a typical example being CCS. We adopt the description method advocated by Plotkin in [13]: an execution step is valid only if it may be *proved* by means of some inference rules. This holds for all the transition relations we shall introduce.

In dealing with concurrency we follow Milner's idea ([11], *cf.* also [6]): a program $(p \parallel q)$ may perform composite actions, product of actions of p and q; unlike SCCS however we assume an asynchronous parallel composition, as it was used in MEIJE([1,4]). Moreover since actions are merely elementary programs there will be no distinction between product and parallel composition: the product of a and b is $(a \| b)$; this is another departure from SCCS. Then for instance the program $(a \parallel b)$ performs itself as an action and terminates; it also performs a, then b, or b, then a, still terminating in doing so.

Usually (operational) semantics aims at describing the behaviour of systems made out of a program p operating on data stored in a memory μ, *cf.* [13]. Such a pair, denoted $(p\backslash\mu)$, is a *configuration* of some "abstract machine". The semantics of sequential languages describes how configurations evolve up to terminal ones. When dealing with sequential programs one is in fact interested in (unlabelled) sequences of transitions of the form $(p\backslash\mu) \to^* \mu'$, which are then abstracted into an input/output behaviour $p\colon \mu \mapsto \mu'$. However these transitions may be built up step by step, so that

$$\mathit{environment} \quad \vdash \quad (p\backslash\mu) \xrightarrow{a} (p'\backslash\mu')$$

holds when the program p performs the action a and becomes p' in doing so, while a operates on the memory μ with μ' as a result. We shall regard this transition relation as defining the (dynamic) semantics of our language. As we can see, a proof of such a step is really twofold: it calls upon the (formal) execution of the program, and upon another kind of behaviour, according to which a program may *operate* on data stored in a memory. This is formalized by means of another transition relation:

$$\mathit{environment} \quad \vdash \quad \mathit{memory} \xrightarrow[\mathsf{oper}]{\mathit{program}} \mathit{memory}'$$

The oper procedure may use the exec one, for the operation of a program is generally that of its *terminated sequences* of actions. For instance the operation of $a\,;b$ is that of a followed by that of b. It is easy to imagine how assignments operate on the memory. At the deepest level, which is that of (guarded) assignments, we shall postulate *mutual exclusion*: assignments are uninterruptible and mutually exclusive atomic operations; then concurrent assignments operate as their nondeterministic interleaving. However we cannot assume that interleaving is the only way to compute the operation of any concurrent program. This is because we want to be able to build configurations which can perform a compound action $(\alpha \parallel \alpha^{-1})$, but cannot perform just α or α^{-1}. The exec procedure cannot insure this mutual inclusion: since the execution of a $(p \parallel q)$ is an "asynchronous" one, a program performing an $(a \parallel b)$ can always perform also a and b. Then the rendez-vous will be achieved by abstracting a co-operation, as we shall see now.

In order to introduce higher levels in the language we use *definitional mechanisms*, that the environment will incorporate. The semantics of such constructs consists in substituting what is defined by its definition ("body", or "code"). In the proposed semantical formalism there are two positions in which one may substitute the definition for a program:

- the first one is what is at the left of the exec arrow. In this place one may introduce a *process* whose execution is that of its body;
- the second position is what is over the oper arrow; here one may introduce what we shall call an *atom*.

Then we have two new instructions in the language: process calls and atom calls. The latter also are actions, and therefore may appear over the exec arrow. The semantics of atoms is such that an atom *operates* as its code, thus possibly in many elementary steps, but is *performed* atomically: a program which performs an atom call will never have to execute its code. This is the way we introduce *levels of abstraction*. Within the scope of its definition an atom can be used as a new primitive uninterruptible operation.

It is worth pointing out that the distinction between processes and atoms (that is between exec and oper) is irrelevant in sequential languages – as a matter of fact, sequential languages stay at the oper side, while CCS-like languages stay at the exec one. Since the semantics of a sequential program is its input/output behaviour, the program ("process", or more accurately procedure) and the abstract function ("atom") it computes are semantically interchangeable in any sequential context. Obviously this is no longer true for parallel programs which communicate during their computations. A first point is that non-terminating programs are perfectly meaningful, such as (while *true* do $x{:=}x+1$); but the main point is that concurrent programs sharing something may interrupt each other. For instance it is very different to regard $(x{:=}x+1\,;x{:=}x+1)$ as an atom (equivalent to $x{:=}x+2$) or as a process: they are not interchangeable in the context $(\ldots \| \, x{:=}2x)$.

In our proposal atoms are part of the (*communication*) *structure* definitions. In fact such structures are much like abstract data types or monitors: they consist of a collection of object types and atoms (there should also be functions). The atoms share objects which represent a local structured memory. Since we want to achieve synchronization (e.g. rendez-vous) by means of such structures, there is no special discipline for atom calls; this makes a difference from monitors, and from the concept of atomic action too, for we do not assume serializability (*cf.* [10] and [9] where one can find more references). On the other hand atoms are "recoverable": they operate in an "all-or-nothing" manner; this is written down in the semantics.

To sum up, let us explain how the whole proposed framework deals with CCS synchronization (value passing will be treated below). We define the CCS port structure as a record-like type with two boolean fields, provided with two atoms for sending and receiving:

$$\begin{aligned} &\mathit{port} = (b{:}\,\mathsf{bool}{=}\mathit{tt} \ \mathsf{and}\ b'{:}\,\mathsf{bool}{=}\mathit{tt}) \\ &\mathsf{with} \\ \mathit{send}(o{:}\,\mathit{port}) &= (\mathsf{when}\ o.b\ \mathsf{do}\ o.b{:=}\mathit{ff})\,;(\mathsf{when}\ \neg o.b'\ \mathsf{do}\ o.b'{:=}\mathit{tt}) = s\,;s' \\ \mathit{receive}(o{:}\,\mathit{port}) &= (\mathsf{when}\ o.b'\ \mathsf{do}\ o.b'{:=}\mathit{ff})\,;(\mathsf{when}\ \neg o.b\ \mathsf{do}\ o.b{:=}\mathit{tt}) = r\,;r' \end{aligned}$$

Let μ be a memory containing a variable u of type *port*, and let p be a program which performs (exec) an action a which is either $\alpha{=}\mathit{send}(u)$, or $\alpha^{-1}{=}\mathit{receive}(u)$, or the parallel product $(\alpha\|\alpha^{-1})$. In order to determine the behaviour of the configuration $(p\backslash\mu)$, in the context of an environment which contains the *port* definition, we have to prove that the action a operates on μ. Since a is made out of atoms, the only means is to unfold these atoms into their codes, while simultaneously unfolding the memory into a "concrete" one μ', containing $u.b = \mathit{tt}, u.b' = \mathit{tt}$. Then we have to prove that the program q, which is either $s\,;s'$, or $r\,;r'$, or $(s\,;s' \| \, r\,;r')$ operates on μ'. Clearly in the first two cases only the first instruction (s or r) may operate, not the second one. Therefore q does not operate on μ', since oper holds only up to the completion of the program. The configuration $(p\backslash\mu)$ cannot perform *send*(u) or *receive*(u): there is no proof of such a transition. On the other hand we have

$$\left\{\begin{matrix} u.b = \mathit{tt} \\ u.b' = \mathit{tt} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{s} \left\{\begin{matrix} u.b = \mathit{ff} \\ u.b' = \mathit{tt} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{r} \left\{\begin{matrix} u.b = \mathit{ff} \\ u.b' = \mathit{ff} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{s'} \left\{\begin{matrix} u.b = \mathit{tt} \\ u.b' = \mathit{ff} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{r'} \left\{\begin{matrix} u.b = \mathit{tt} \\ u.b' = \mathit{tt} \end{matrix}\right\}$$

Since $s\ r\ s'\ r'$ is a terminated sequence of actions of $(s\,;s' \| \, r\,;r')$, we have a proof that $(p\backslash\mu)$ may perform $(\alpha \, \| \, \alpha^{-1})$.

NOTE. The concept of composite actions seems to be crucial here: could we "implement" the rendez-vous by means of a purely sequential non-deterministic interpretation of concurrency? The idea that an atom abstracts the terminated behaviours of its code has been used in dealing with semantics and verification of concurrent systems ([4,5]); there it led to the notions of observation criterion and transition systems morphism. One may regard the abstraction mechanism of our language as embodying a syntactical (operational) version of these notions.

2. Data and Programs.

Operational semantics aims at describing the behaviour of systems made from a program and data. In this section we shall begin with a few words about data, and then give the syntax of programs. We shall assume a collection of data types – among which the booleans (bool), the integers (int), and so on. Associated with each type are expressions and values. For instance the logical constants tt and ff are expressions, and one may form other expressions using functions like negation $(\neg b)$ or conjunction $(b \wedge b')$. Thus we assume an underlying language of *typed expressions* – such as the applicative part of ML, or something more general, *cf.* [7].

In order to write expressions, we also need identifiers. We therefore assume a denumerable set Ident of *identifiers*, for instance the set of non-empty words written on a finite alphabet Alph:

$$\mathsf{Ident} = (\mathsf{Alph})^+$$

We shall use record-like types, thence we allow compound variables like $x.y$ to be expressions. Let "." be a symbol not in Alph; then the set of *variables* is given by

$$\mathsf{Var} = \mathsf{Ident}(.\mathsf{Ident})^*$$

In the sequel we let $x, y, z \ldots$ range over identifiers and $u, v, w \ldots$ over variables.

We must now define the *record types* we use – there may be some differences with the usual (PASCAL or ML) notions of record (†). Our record types may be recursively defined, so that fields of a record are determined accordingly. If for instance

$$complex = (r\colon \mathsf{real}\ \mathsf{and}\ i\colon \mathsf{real})$$

then a *complex* is a record with two real fields r and i, and if c is of type *complex* then $c.r$ and $c.i$ are variables of type real.

The syntax of our language is the following:

RECORD TYPES:

(i) if x is an identifier, τ a type (from the expression language), and e an expression of that type, then $x\colon\tau{=}e$ is a record type. Its only field is x;

(ii) if x is an identifier, ω is a type or a record type, then $x\colon\omega$ is a record type. Its only field is x;

(iii) if ω and ω' are record types whose fields are distincts then $(\omega\ \mathsf{and}\ \omega')$ is a record type whose fields are those of ω and of ω';

(iv) if π is an identifier and $\omega_1, \ldots, \omega_n$ are record types, types, or identifiers, then $\pi(\omega_1, \ldots, \omega_n)$ is a record type (whose fields are those of the definition of π).

NOTE. We shall give later the syntax of record type definition. We have not given any means to form *expressions* of record types, hence only variables may have such a type, and there is no expressible *value* of record types.

ACTIONS:

(i) if $v_1, \ldots, v_n$ are distinct variables and $e_1, \ldots, e_n$ are expressions then the *assignment* $(v_1, \ldots, v_n){:=}(e_1, \ldots, e_n)$ is an action;

(ii) if t is a boolean expression and a an action then the *boolean guard* (when t do a) is an action;

(†) In case the underlying language has record types, we might use another word, such as "profile" or "store".

(*iii*) if a and b are actions then their *parallel composition* $(a \parallel b)$ is an action;
(*iv*) if a is an *atom call* then it is an action.
(*v*) if d is a *definition* and a an action then $(\mathsf{def}\ d\ \mathsf{in}\ a)$ is an action.

We shall give later the syntax of definitions and atom calls. We do not say anything about *static semantics*, that is about what should be needed to check that assigned variables are distinct, that expressions have an appropriate type, that the guarding expression t is of boolean type, and so on (*cf.* [13]).

PROGRAMS:
(*i*) any action is a program;
(*ii*) if t is a boolean expression and p a program then the *boolean guard* $(\mathsf{when}\ t\ \mathsf{do}\ p)$ is a program;
(*iii*) if p and q are two programs then their *sequential* and *parallel composition* $p\,;q$ and $(p \parallel q)$ are programs;
(*iv*) if p is a *process call* then it is a program;
(*v*) if d is a *definition* and p a program then $(\mathsf{def}\ d\ \mathsf{in}\ p)$ is a program.

The syntax of process calls will be given in a next section. We could also add *conditionals* of the form $(\mathsf{if}\ t\ \mathsf{then}\ p\ \mathsf{else}\ q)$. In the following we let $a, b, c\ldots$ range over actions and $p, q, r\ldots$ over programs.

3. Execution of Elementary Programs.

As indicated in the introduction, programs perform actions in a single step. The method to describe these transition steps is that advocated by Plotkin in [13]: they are the ones we may *prove* according to a set of *inference rules* and *axioms*.

The first element of this specification set is an axiom which says that in any environment ε an assignment a is a process which performs itself – as an action – and terminates. There is nothing in the syntax which is intended to mean "termination": there is no "skip" program. Thus we need a special symbol $\mathbb{1}$ which is *not* a program but may be the result of an execution. Then our first statement, where a is an assignment $(v_1, \ldots, v_n){:=}(e_1, \ldots, e_n)$, is:

$$\boxed{\textit{assignment (execution)}: \quad \varepsilon \vdash a \xrightarrow[\text{exec}]{a} \mathbb{1}}$$

This axiom indicates that an assignment is *uninterruptible* since its execution cannot be broken into many steps.

The execution of a guard is specified in a simple manner: $(\mathsf{when}\ t\ \mathsf{do}\ p)$ performs the same actions as p but formally guarded by the test – here we do not have to take care whether the test is true or not. This is formalized by the rule:

$$\boxed{\textit{guard (execution)}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p'}{\varepsilon \vdash (\mathsf{when}\ t\ \mathsf{do}\ p) \xrightarrow[\text{exec}]{(\mathsf{when}\ t\ \mathsf{do}\ a)} p'}}$$

REMARKS.
(*1*) When p terminates, i.e. when $p' = \mathbb{1}$ then the guard $(\mathsf{when}\ t\ \mathsf{do}\ p)$ terminates.
(*2*) The boolean test t only guards the initial actions of the program p, since the resulting program p' is no longer guarded.

In the same manner we could describe the execution of a conditional, if we regard it as a "sum"

$$(\mathsf{if}\ t\ \mathsf{then}\ p\ \mathsf{else}\ q) \equiv (\mathsf{when}\ t\ \mathsf{do}\ p) + (\mathsf{when}\ \neg t\ \mathsf{do}\ q)$$

We do *not* introduce a sum or "select" construct in the language, as it is done in CSP or CCS; however this suggests the following rules for the execution of a conditional:

$$\textit{conditional 1}\ : \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p'}{\varepsilon \vdash (\text{if } t \text{ then } p \text{ else } q) \xrightarrow[\text{exec}]{(\text{when } t \text{ do } a)} p'}$$

$$\textit{conditional 2}\ : \quad \frac{\varepsilon \vdash q \xrightarrow[\text{exec}]{b} q'}{\varepsilon \vdash (\text{if } t \text{ then } p \text{ else } q) \xrightarrow[\text{exec}]{(\text{when } \neg t \text{ do } b)} q'}$$

We could also introduce other conditional constructs, such as (while t do p) or (if t then p) which terminate when the test is false. These will not be needed for our immediate purpose. Moreover, in order to describe their semantics, we should have to regard $\mathbb{1}$ as an action – thus also a program, as in [6] –, stating something like

$$(\text{if } t \text{ then } p) \xrightarrow[\text{exec}]{(\text{when } \neg t \text{ do } \mathbb{1})} \mathbb{1}$$

There is no surprise for what regards the execution of a sequence $p\,;q$, which is determined by two rules. In any case such a program first performs an action of p, and what it becomes depends whether p terminates or not:

$$\boxed{\textit{sequential composition 1}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p' \neq \mathbb{1}}{\varepsilon \vdash p\,;q \xrightarrow[\text{exec}]{a} p'\,;q}}$$

In this rule $p' \neq \mathbb{1}$ is a purely syntactical test, as it will be always the case in what follows; this does not mean testing deadlock or termination of an algorithm!

$$\boxed{\textit{sequential composition 2}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} \mathbb{1}}{\varepsilon \vdash p\,;q \xrightarrow[\text{exec}]{a} q}}$$

NOTE. We could have introduced sequential composition in the actions – as it is done in [6] – by a rule like

$$p \xrightarrow[\text{exec}]{a} \mathbb{1}\ ,\ q \xrightarrow[\text{exec}]{b} q' \quad \Rightarrow \quad p\,;q \xrightarrow[\text{exec}]{a\,;b} q'$$

Such composite actions could be required in the formalization of the implementation of the ESTEREL programming language ([2]), for the semantics of this language uses "instantaneous" actions involving causal dependencies.

The semantics of a parallel composition is, roughly speaking, that of MEIJE [1,4], or more accurately that of [6]. Let us assume for a while as in [6] that $\mathbb{1}$ is an action that any program may perform without any state change ($p \xrightarrow[\text{exec}]{\mathbb{1}} p$). Then the execution of $(p \parallel q)$ is defined by the single rule

$$p \xrightarrow{a} p' \; , \; q \xrightarrow{b} q' \;\Rightarrow\; (p \parallel q) \xrightarrow{(a \parallel b)} (p' \parallel q')$$

Since $(a \parallel \mathbb{1}) = a = (\mathbb{1} \parallel a)$ (in the formalism of [6]) this is more or less the asynchronous parallel composition of MEIJE, where an action of $(p \parallel q)$ is an action of one of its components, or results from their simultaneous activity. Moreover we have to take into account termination ($p' = \mathbb{1}$ or $q' = \mathbb{1}$); therefore in our formalism where we want to avoid the use of semantical laws such as $(x \parallel \mathbb{1}) = x$ we must describe the execution of $(p \parallel q)$ by means of eight rules. This will be reduced to just four by using non-deterministic rules that entail several possible conclusions. The first rule deals with the case where the activity is of one component only, which terminates:

$$\boxed{\textit{concurrency 1}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} \mathbb{1}}{\varepsilon \vdash (p \parallel q) \xrightarrow[\text{exec}]{a} q \quad , \quad \varepsilon \vdash (q \parallel p) \xrightarrow[\text{exec}]{a} q}}$$

The following is the same, but the component does not terminate:

$$\boxed{\textit{concurrency 2}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p' \neq \mathbb{1}}{\varepsilon \vdash (p \parallel q) \xrightarrow[\text{exec}]{a} (p' \parallel q) \quad , \quad \varepsilon \vdash (q \parallel p) \xrightarrow[\text{exec}]{a} (q \parallel p')}}$$

When the action is a compound one, and at least one of the components terminates, one has:

$$\boxed{\textit{concurrency 3}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} \mathbb{1} \quad , \quad \varepsilon \vdash q \xrightarrow[\text{exec}]{b} q'}{\varepsilon \vdash (p \parallel q) \xrightarrow[\text{exec}]{(a \parallel b)} q' \quad , \quad \varepsilon \vdash (q \parallel p) \xrightarrow[\text{exec}]{(b \parallel a)} q'}}$$

Finally there is the case where the two components perform something concurrently, and neither terminates:

$$\boxed{\textit{concurrency 4}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p' \neq \mathbb{1} \quad , \quad \varepsilon \vdash q \xrightarrow[\text{exec}]{b} q' \neq \mathbb{1}}{\varepsilon \vdash (p \parallel q) \xrightarrow[\text{exec}]{(a \parallel b)} (p' \parallel q') \quad , \quad \varepsilon \vdash (q \parallel p) \xrightarrow[\text{exec}]{(b \parallel a)} (q' \parallel p')}}$$

REMARKS.

(1) One may note that if the two components terminate, then their parallel composition terminates, and this is the only termination case.

(2) These rules will be the only way to introduce a compound action $(a \| b)$; hence when a programs performs such an action, it may also perform a and b.

This ends the execution rules of elementary programs; the semantics of the other constructs – atom calls, process calls, definitions – will be given below.

4. Operation of Elementary Programs.

In this section we shall describe how simple programs use and modify data, that is we shall describe the oper procedure. The corresponding transition relation, as well as the previous relation exec, is needed to synthesize the behaviour of *configurations*, that are pairs of programs and data.

4.1 The Memory.

Data are stored in a *memory*, which is a partial mapping from a (finite non-empty) set of variables to values – this mapping is partial since some variables may have been *declared* but not initialized. It will be convenient to add some information about types, so that a memory can be represented as

$$\{v_1\!:\tau_1{=}\mathrm{v}_1,\ldots,v_n\!:\tau_n{=}\mathrm{v}_n,u_1\!:\omega_1,\ldots,u_k\!:\omega_k\}$$

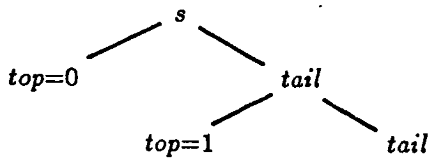
if $\{v_1,\ldots,v_n,u_1,\ldots,u_k\}$ is the set of declared variables, $\mathrm{v}_1,\ldots,\mathrm{v}_n$ are the respective values of the v_i's, and the τ_i's and ω_j's are the types. Since we have introduced record types we must say something about the value of a variable u having such a type. With respect to a memory μ, this value is just another memory, part of the given one. For instance let $stack(\tau)$ be the recursively defined record type of *pushdown stacks* (of items of type τ)

$$stack(\tau) = (top\!:\tau \text{ and } tail\!:stack(\tau))$$

and let s be an identifier of type $stack(\mathsf{int})$. Then, given the memory

$$\mu = \{s.top\!:\mathsf{int}{=}0\ ,\ s.tail.top\!:\mathsf{int}{=}1\ ,\ s.tail.tail\!:stack(\mathsf{int})\}$$

(where $s.tail.tail$ has no value), the value of $s.tail$ is the "sub-memory" $\{top\!:\mathsf{int}{=}1, tail\!:stack(\mathsf{int})\}$. Therefore the convenient model for a memory is that of a set of *trees* with typed and possibly valued leaves; nodes are labelled by identifiers, so that variables are paths in the trees. The memory μ above may thus be drawn, omitting the types



Formally a memory μ is then a triple (M,ν,θ) such that

(1) $M \subseteq \mathsf{Var}$ is the finite non-empty set of *paths*, satisfying $u.v \in M \Rightarrow u \in M$;

(2) ν is a partial mapping from the set $l(M) = \{u/u \in M \;\&\; \forall v\; u.v \notin M\}$ of *leaves* to values of underlying types (bool, int, and so on);

(3) θ is the mapping assigning types to (all) the leaves.

Although we require values to be of underlying types, and not of record types, we allow as a *notational convenience* v_i to be a memory in

$$\{v_1\!:\tau_1{=}\mathrm{v}_1,\ldots,v_n\!:\tau_n{=}\mathrm{v}_n,u_1\!:\omega_1,\ldots,u_k\!:\omega_k\}$$

For instance the μ of the previous example could be denoted

$$\{s\!:stack(\mathsf{int}){=}\{top\!:\mathsf{int}{=}0\ ,\ tail\!:stack(\mathsf{int}){=}\{top\!:\mathsf{int}{=}1\ ,\ tail\!:stack(\mathsf{int})\}\}\}$$

Moreover the type information will be omitted in most examples. This representation shows that a memory is just a special kind of Kahn-Plotkin's *concrete data structure*, *cf.* [3].

Given $\mu = (M, \nu, \theta)$, the value of a variable $u \in M - l(M)$ (a leaf $v \in l(M)$ obviously has value $\nu(v)$) is

$$\mu(u) = (M/u, \nu/u, \theta/u) \quad \text{where} \quad \left\{ \begin{array}{ll} M/u = \{v/u.v \in M\} & \text{and for } v \in l(M/u) \\ (\theta/u)(v) = \theta(u.v) & \text{and} \\ (\nu/u)(v) = \nu(u.v) & \text{if } u.v \in \mathrm{dom}(\nu) \\ & \text{undefined otherwise} \end{array} \right.$$

Programs operate on the memory by *updating* it, assigning a new value to some variables. We define this as the modification of an old memory $\mu = (M, \nu, \theta)$ by another one $\mu' = (M', \nu', \theta')$ resulting in the new $\mu[\mu'] = (M[M'], \nu[\nu'], \theta[\theta'])$. The set $M[M']$ of new paths is that of M', plus the paths of M which are not suffixes of a leaf of M':

$$\begin{array}{rl} M[M'] & = M' \cup (M - l(M')(.\mathsf{Ident})^*) \\ \nu[\nu'] & = \left\{ \begin{array}{ll} \nu'(u) & \text{if } u \in \mathrm{dom}(\nu') \\ \nu(u) & \text{if } u \in \mathrm{dom}(\nu) - l(M')(.\mathsf{Ident})^* \\ & \text{undefined otherwise} \end{array} \right. \\ \theta[\theta'] & = \left\{ \begin{array}{ll} \theta'(u) & \text{if } u \in l(M') \\ \theta(u) & \text{if } u \in l(M) - l(M')(.\mathsf{Ident})^* \end{array} \right. \end{array}$$

Obviously to compute the operation of assignments and guards one must use an *evaluation mechanism* for the expressions, by means of which one computes the value of, say, e in the context of a memory μ. This will be denoted

$$\langle \varepsilon \, , \, \mu \rangle \vdash e \xrightarrow[\mathsf{eval}]{} \mathrm{v}$$

so that for instance $\mathrm{v} = \mu(e)$ if e has a record type (in which case e ought to be a variable). Here the environment may contain definitions of functions. We do not say anything about the evaluation of expressions of the presumed underlying language, and for more details we refer to [13]; however we make a single assumption, that evaluation is *purely applicative* – this is in fact implicit in the previous notation. In our formalism the non-existence of side effects means that the eval procedure never calls the oper one.

4.2 Assignments and Guards.

The whole technical apparatus of the previous section allows us to formalize the operation of an assignment

$$a = (v_1, \ldots, v_n) {:=} (e_1, \ldots, e_n)$$

Intuitively this is rather simple: one evaluates the expressions e_i and accordingly updates the v_i. The rule is as follows:

$$\boxed{\textit{assignment (operation)}: \quad \frac{\langle \varepsilon \, , \, \mu \rangle \vdash e_1 \xrightarrow[\mathsf{eval}]{} \mathrm{v}_1 \, , \ldots , \, \langle \varepsilon \, , \, \mu \rangle \vdash e_n \xrightarrow[\mathsf{eval}]{} \mathrm{v}_n}{\varepsilon \vdash \mu \xrightarrow[\mathsf{oper}]{a} \mu[\{v_1{=}\mathrm{v}_1 \, , \ldots , \, v_n{=}\mathrm{v}_n\}]}}$$

In this rule we use the previous notational convention for $\{v_1{=}\mathrm{v}_1 \, , \ldots , \, v_n{=}\mathrm{v}_n\}$. Continuing the above example, the assignment $(s.top, s.tail){:=}(s.top + 1, s)$ updates the memory

$$\mu = \{s.top{=}0 \ , \ s.tail.top{=}1 \ , \ s.tail.tail\}$$

into

$$\{s.top{=}1 \ , \ s.tail.top{=}0 \ , \ s.tail.tail.top{=}1 \ , \ s.tail.tail.tail\}$$

REMARKS. It is worth emphasizing two facts about this rule:

(1) first we regard it as an *axiom* of our formalism: this is right because we have assumed a purely applicative evaluation of expressions. Therefore the inference bar of this rule makes a strong parting with the underlying (functional) language. Moreover this means that assignments operate in an *indivisible* manner since there are no oper steps during the evaluation, as there are no steps in updating.

(2) we must say that this rule will be the *only* axiom of the oper procedure. Thus in order to prove an oper transition we will always have to come down to the operation of assignments. These deepest instructions are in fact the only ones by which a program may operate on data, even if this can be somehow hidden (by the abstraction mechanism).

For all these reasons we regard assignments as the *particles* of the language – this is true also with respect to the exec relation.

Boolean guards operate as the guarded program when the test is true, that is:

$$\boxed{\textit{guard (operation)}: \quad \frac{\langle \varepsilon \,,\, \mu \rangle \;\vdash\; t \xrightarrow[\text{eval}]{} tt \quad , \quad \varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{p} \mu'}{\varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{(\text{when } t \text{ do } p)} \mu'}}$$

This is the single rule for guard's operation; thus, when the test is false (or unevaluable) the guard cannot operate. One should also note that this time the entire operation, and not only its first step, is guarded by the test. From this point of view an action such as

$$(\text{when } t \text{ do } (v_1, \ldots, v_n){:=}(e_1, \ldots, e_n))$$

is like a generalized *test-and-set*, as for instance the Dijkstra's P and V operations on semaphores.

4.3 Sequences of Operations.

When introducing the oper procedure, we have said that the operation of a program is that of its terminated sequences of actions. Two general rules formalize this idea – according to which our notion of operation is very much like the usual mathematical one (operation of a semi-group on a set). A program must perform at least one action to complete a sequence, and the first rule deals with this "immediate completion":

$$\boxed{\textit{operation (completion)}: \quad \frac{\varepsilon \;\vdash\; p \xrightarrow[\text{exec}]{a} \mathbb{1} \quad , \quad \varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{a} \mu'}{\varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{p} \mu'}}$$

The next rule states that operation may result from a sequence:

$$\boxed{\textit{operation (sequence)}: \quad \frac{\varepsilon \;\vdash\; p \xrightarrow[\text{exec}]{a} p' \quad , \quad \varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{a} \mu'' \xrightarrow[\text{oper}]{p'} \mu'}{\varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{p} \mu'}}$$

In this rule $\varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{a} \mu'' \xrightarrow[\text{oper}]{p'} \mu'$ is an abbreviation for $\varepsilon \;\vdash\; \mu \xrightarrow[\text{oper}]{a} \mu''$, $\varepsilon \;\vdash\; \mu'' \xrightarrow[\text{oper}]{p'} \mu'$.

Naturally enough these rules allow us to infer the operation of a sequential composition $p\,;q$, as it is examplified in the following proof (where we omit the irrelevant environment):

$$\frac{\dfrac{x{:=}x+1 \xrightarrow[\text{exec}]{x:=x+1} \mathbb{1}}{(x{:=}x+1)\,;(x{:=}2x) \xrightarrow[\text{exec}]{x:=x+1} x{:=}2x} \qquad \dfrac{\cdots}{\{x{=}0\} \xrightarrow[\text{oper}]{x:=x+1} \{x{=}1\} \xrightarrow[\text{oper}]{x:=2x} \{x{=}2\}}}{\{x{=}0\} \xrightarrow[\text{oper}]{(x:=x+1)\,;(x:=2x)} \{x{=}2\}}$$

It is then unnecessary to state any inference rule for the operation of sequential composition $p\,;q$ – note that $p\,;q$ is an interruptible program for both exec and oper. What is much more important is that we do not state any specific inference rule for parallel composition. The consequence is that we still have to use the general rules above to determine the operation of $(p \parallel q)$ – at least this is true for elementary programs (without abstraction). For instance let

$$p = (a \parallel b) = (x{:=}x+1 \parallel x{:=}2x)$$

The terminated executions of this program are, omitting the environment:

$$p \xrightarrow[\text{exec}]{x:=x+1} (x{:=}2x) \xrightarrow[\text{exec}]{x:=2x} \mathbb{1} \quad , \quad p \xrightarrow[\text{exec}]{x:=2x} (x{:=}x+1) \xrightarrow[\text{exec}]{x:=x+1} \mathbb{1} \quad , \quad p \xrightarrow[\text{exec}]{(a \parallel b)} \mathbb{1}$$

Now we want to know how p operates on the memory $\{x{=}0\}$. We find out that the only possible proofs are instances of the previous rule, e.g.

$$(a \parallel b) \xrightarrow[\text{exec}]{a} b \quad , \quad \mu \xrightarrow[\text{oper}]{a} \mu'' \xrightarrow[\text{oper}]{b} \mu' \quad \Rightarrow \quad \mu \xrightarrow[\text{oper}]{(a \parallel b)} \mu'$$

Then we get either $\{x{=}1\}$ or $\{x{=}2\}$. One must conclude that the operation of a concurrent program $(p \parallel q)$ is that of the non-deterministic interleaving of its components – again this is true for elementary programs. The absence of an inference rule for concurrent operations formalizes the postulate that assignments are mutually exclusive operations. As an exercise, the reader could establish that the programs $((x{:=}x+1)\,;(x{:=}x+1) \parallel x{:=}2x)$ and $(x{:=}x+2 \parallel x{:=}2x)$ do not have the same operation.

To conclude this section we give the semantics of configurations. First let us give the syntax of this new species.

CONFIGURATIONS:
(i) if μ is a memory then $(\mathbb{1}\backslash\mu)$ is a configuration;
(ii) if μ is a memory and p a program then $(p\backslash\mu)$ is a configuration.
Then configurations perform some transitions that we regard as defining the operational semantics of our language – we do not qualify this new kind of transition (although the qualification might be exec). However, given an environment ε, only *closed* configurations are allowed to perform actions; a configuration $(r\backslash\mu)$ is closed in ε if any variable occurring free in r is either defined (in ε) or declared (in μ) – this should be checked by static semantics, *cf.* [13]. The following rule states that a possible transition of a configuration $(p\backslash\mu)$ results from both the activity of the program p and the operation of this activity on the memory μ:

$$\boxed{\textit{(closed) configurations}: \quad \frac{\varepsilon \vdash p \xrightarrow[\text{exec}]{a} p' \;,\; \varepsilon \vdash \mu \xrightarrow[\text{oper}]{a} \mu'}{\varepsilon \vdash (p\backslash\mu) \xrightarrow{a} (p'\backslash\mu')}}$$

REMARK. At this point we may explain why exec is a "symbolic" procedure: in order to prove an execution step of a program, one has only to use the execution of subprograms, but no operation on the data. On the other hand the oper procedure may require the exec one: this is apparent in the two general rules above. Moreover we must point out that, as a consequence of the given rules, the operation of a program is only determined by its *terminated* sequences of actions.

5. Processes and Atoms.

5.1 Execution and Operation of Blocks.

Definitions have, in the construct $(\mathsf{def}\ d\ \mathsf{in}\ p)$, a *local scope*; we thence call *block* such a construct. We shall assume that the environment is a pushdown stack of definitions, and denote $(\varepsilon\frown d)$ the stack we get by pushing d on ε. Then the execution of a block $(\mathsf{def}\ d\ \mathsf{in}\ p)$ is that of p in the environment enriched with d. This is formalized by the two following rules:

$$\boxed{\textit{block (completion)}:\quad \frac{(\varepsilon\frown d)\ \vdash\ p \xrightarrow[\mathsf{exec}]{a} \mathbb{1}}{\varepsilon\ \vdash\ (\mathsf{def}\ d\ \mathsf{in}\ p) \xrightarrow[\mathsf{exec}]{(\mathsf{def}\ d\ \mathsf{in}\ a)} \mathbb{1}}}$$

Note that a definition vanishes at completion; on the other hand one has:

$$\boxed{\textit{block (execution)}:\quad \frac{(\varepsilon\frown d)\ \vdash\ p \xrightarrow[\mathsf{exec}]{a} p' \neq \mathbb{1}}{\varepsilon\ \vdash\ (\mathsf{def}\ d\ \mathsf{in}\ p) \xrightarrow[\mathsf{exec}]{(\mathsf{def}\ d\ \mathsf{in}\ a)} (\mathsf{def}\ d\ \mathsf{in}\ p')}}$$

These rules introduce the definition as a part of the action. This is needed for the following reason: in order to prove a transition of a configuration $((\mathsf{def}\ d\ \mathsf{in}\ p)\backslash\mu)$ one has to determine the operation of some action of the program $(\mathsf{def}\ d\ \mathsf{in}\ p)$ on μ, and this may require knowledge of some definitions (of atoms, see below). Actions of the form $(\mathsf{def}\ d\ \mathsf{in}\ a)$ operate in an obvious manner:

$$\boxed{\textit{block (operation)}:\quad \frac{(\varepsilon\frown d)\ \vdash\ \mu \xrightarrow[\mathsf{oper}]{a} \mu'}{\varepsilon\ \vdash\ \mu \xrightarrow[\mathsf{oper}]{(\mathsf{def}\ d\ \mathsf{in}\ a)} \mu'}}$$

There is no specific rule to determine the operation of a general block – $(\mathsf{def}\ d\ \mathsf{in}\ p)$ where p is not an action –, which therefore must be proved by means of the previous general rules (sequences of operations).

5.2 Recursive Processes.

A (recursive) process is not generally intended to terminate and provide an output, but rather to proceed endlessly; in its course it updates the memory. Since we want to avoid "side effects" the definition of a process explicitly marks what variables it uses, and how they are used. Intuitively there are three uses of variables: a program *imports* a variable when it needs to reach its value in the memory, while it *exports* the variables it updates – the program may also use *objects*, by means of atom calls; we shall see that later. The variables imported by the body of a procedure will be formal parameters marked !, since in a procedure call the corresponding actual parameters are expressions "sent" to the body. Conversely, a parameter is marked ? when it is updated by the body. The syntax is as follows: a (*recursive*) *definition of processes* is a system of equations

$$\{\varphi_i(z_1,\ldots,z_{m_i})(!x_1,\ldots,x_{n_i})(?y_1,\ldots,y_{k_i}) = q_i /\ 1 \leq i \leq \ell\}$$

where

(1) the φ_j's (the names of the defined processes), z_j's, x_j's and y_j's (the formal parameters) are distinct identifiers, and the q_j's are programs;

(2) the object, imported, and exported variables occurring free in q_i are among $z_1,\ldots,z_{m_i}$, $x_1,\ldots,x_{n_i}$, and $y_1,\ldots,y_{k_i}$ respectively.

Static semantics might formalize the correct use of variables; this is deferred to a subsequent paper. In a more concrete syntax we should write a process definition proc(d_1 and $\cdots$ and d_ℓ), where each d_i is an equation. Moreover it would be worthwhile to enrich the syntax - e.g. with "sequential" constructs such as inside and enclose, *cf.* [13]. We do not spend much care on the notion of recursive process, which is a rather standard one. The only thing to say is that a process definition allows to use *process calls*, that is, programs of the form

$$p = \varphi(u_1,\ldots,u_m)(e_1,\ldots,e_n)(v_1,\ldots,v_k)$$

where the e_j's are expressions, and the u_j' s and v_j's are variables (the v_j's being distinct). Such a process call exports the v_j's, while it imports the variables occurring in the e_j's, and uses the objects u_j's. For instance

$$(\mathsf{def}\ \ \varphi()(!x)(?y) = (y{:=}x+1)\,;\varphi()(x)(y)\ \ \mathsf{in}\ \ z{:=}0\,;\varphi()(z)(z))$$

is a syntactically correct program. This program yields the sequence of integers on the variable z.

In order to execute a process call $\varphi\langle args\rangle = \varphi(u_1,\ldots,u_m)(e_1,\ldots,e_n)(v_1,\ldots,v_k)$, one has to search the definition of φ in the environment ε - let us denote it $\mathsf{def}(\varphi,\varepsilon)$. Note that since the environment is a stack, the first encountered definition is the right one. Then the execution of $\varphi\langle args\rangle$ is that of $\mathsf{def}(\varphi,\varepsilon)$ applied to the appropriate arguments. If we denote

$$\mathsf{def}(\varphi,\varepsilon)\langle args\rangle = \mathsf{def}(\varphi,\varepsilon)\,[u_1/z_1,\ldots,u_m/z_m,e_1/x_1,\ldots,e_n/x_n,v_1/z_1,\ldots,v_k/z_k]$$

(in our formalism substitution is the parameter passing mechanism), then the rule is

$$\boxed{\textit{process call}: \quad \frac{\varepsilon \ \vdash\ \mathsf{def}(\varphi,\varepsilon)\langle args\rangle \xrightarrow[\mathsf{exec}]{a} r}{\varepsilon \ \vdash\ \varphi\langle args\rangle \xrightarrow[\mathsf{exec}]{a} r}}$$

Note that process definitions are always (implicitly) recursive since, once used, the definition still remains part of the environment.

5.3 Structures and Atoms.

The last two constructs of our proposal are those of structure definitions and atom calls. A structure is a collection of object types and atoms; objects provide atoms for a shared and structured local memory. We have seen that a memory is denoted by records, thus objects will be records; the syntax of a structure definition is:

$$d = \mathsf{struc}(\textit{record types definition}\ \mathsf{with}\ \textit{atoms definition})$$

The code of atoms is intended to be the only way to use the internal structure of objects, and in fact a structure is very much like an abstract data type. Regarding the data, abstraction is restricted to the simple idea that the "." marks the parting with the "internal structure" (in ML one has *abs* and *rep*; for a more general point of view *cf.* [12,7]. Another more general notion of "abstract" type could be that of *concrete data structure*, *cf.* [3]). The syntax of our (recursive) record types definition is the following:

$$\{\pi_j(\tau_1,\ldots,\tau_{n_j}) = \omega_j /\ 1 \leq j \leq \ell\}$$

where

(1) the π_j's and τ_i's are distinct identifiers, and the ω_j are record types;
(2) the type arguments of an ω_j must be among $\tau_1,\ldots,\tau_{n_j}$;
(3) the π_i's can only occur in the right hand sides as subexpressions $\pi_j(\tau_1,\ldots,\tau_{n_j})$ (the parameters can only be the formal ones);
(4) the π_i's must be *guarded* by a field in the ω_j's: they must occur in a subexpression $x{:}\,\omega$.

This last restriction ensures that the fields of a record type are distinct. For instance something like

$$heap(\tau) = (c{:}\,cell(\tau) \text{ and } heap(\tau))$$

is not a legal definition. However we could lay down a weaker restriction in order to take into account some more general inheritance phenomena (*cf.* [7]) such as

$$\begin{aligned} stack(\tau) &= (top{:}\,\tau \text{ and } tail{:}\,stack(\tau)) \\ pstack(\tau) &= (empty{:}\,\mathsf{bool}{=}tt \text{ and } stack(\tau)) \end{aligned}$$

The syntax of atoms definitions is almost identical to that of procedures definitions; such a definition is a system of equations

$$\{\alpha_i(z_1{:}\,\sigma_1,\ldots,z_{m_i}{:}\,\sigma_{m_i})(!x_1,\ldots,x_{n_i})(?y_1,\ldots,y_{k_i}) = r_i /\ 1 \leq i \leq \ell\}$$

In this definition the object parameters z_i have an explicit type σ_i, which must be one of the $\pi_j(\tau_1,\ldots,\tau_{n_j})$ defined in the structure. The process r_i must use respectively the x_j's and y_j's as imported and exported variables; any other variable occurring free in r_i must be a z_j or a path $z_j.u$ of an object parameter. One can for instance define the (boolean) *semaphore* structure as

$$\begin{aligned} sem = (b{:}\,\mathsf{bool}{=}tt) \text{ with } P(o{:}\,sem) &= (\text{when } o.b \text{ do } o.b{:=}\mathit{ff}) \\ V(o{:}\,sem) &= (\text{when } \neg o.b \text{ do } o.b{:=}tt) \end{aligned}$$

(parentheses which enclose nothing are omitted). A pushdown stack can be defined as the structure:

$$\begin{aligned} stack(\tau) = (top{:}\,\tau \text{ and } tail{:}\,stack(\tau)) \text{ with } push(s{:}\,stack(\tau))(!x) &= (s.top, s.tail){:=}(x, s) \\ pop(s{:}\,stack(\tau))(?y) &= (y, s){:=}(s.top, s.tail) \end{aligned}$$

We shall see some more interesting examples in the next section. One may regard what is at the left of "=" in a structure definition as the specification, or the interface, whereas the right hand side is the implementation: representation of the objects and code of the procedures.

The last program construct is that of *atom calls*, whose syntax is the following:

$$(v_1,\ldots,v_k){:=}\alpha(u_1,\ldots,u_m)(e_1,\ldots,e_n)$$

with the same constraints as for the process calls regarding the v_i's, u_i's and e_i's. We shall simply write $\alpha(u_1,\ldots,u_m)(e_1,\ldots,e_n)$ when $k = 0$. One should note that in almost all "object oriented languages" a class definition introduces only one object type; therefore the procedures (or "methods") have an implicit parameter of that type, and the call (a "message" sent to an object) is qualified by a variable: $u.\alpha(e_1, \ldots, e_n)$.

Contrary to the process calls, an atom call is an action. Let a be such a call; then it is introduced – as an action – by the axiom:

$$\boxed{\textit{atom call (execution)}: \quad \varepsilon \vdash a \xrightarrow[\text{exec}]{a} \mathbb{1}}$$

This and the rule for assignments are the only axioms of the exec transition relation: assignments and atom calls are the uninterruptible atomic actions of our language. One should note that the following is true:

$\forall a$ action $\varepsilon \vdash a \xrightarrow[\text{exec}]{a} \mathbb{1}$ is valid (i.e. may be proved, for some appropriate environment ε)

A program performing an atom call will never execute its code, since no rule of the exec procedure prescribes to unfold such a call. On the other side, the execution of the code is needed to determine the operation of an atom on the memory; this is formalized by the *abstraction* rule. This last rule of the oper procedure is twofold: for one part abstraction relates to data types, for the other it regards programs. In the hypothesis of the rule, one operates on a memory where record types have been unfolded according to the definitions found in the environment; and what operates is the program where atom codes applied to suitable arguments are substituted for atom calls. We shall not enter into the syntactical details, but merely denote $\mu[\mathsf{typ}(\varepsilon)]$ and $p[\mathsf{atom}(\varepsilon)]$ the respective unfoldings of the memory μ and of the program p. For example if $\mu = \{\cdots s\colon \mathit{stack}(\mathsf{int}) \cdots\}$ and ε contains the definition of $\mathit{stack}(\tau)$ above then $\mu[\mathsf{typ}(\varepsilon)]$ will be

$$\mu = \{\cdots s{=}(\mathit{top}\colon \mathsf{int}, \mathit{tail}\colon \mathit{stack}(\mathsf{int})) \cdots\}$$

that is, according to our notational convention, $\mu = \{\cdots s.\mathit{top}\colon \mathsf{int}, s.\mathit{tail}\colon \mathit{stack}(\mathsf{int})) \cdots\}$. The announced rule is the following:

$$\boxed{\textit{abstraction}: \quad \dfrac{\varepsilon \vdash \mu[\mathsf{typ}(\varepsilon)] \xrightarrow[\text{oper}]{p[\mathsf{atom}(\varepsilon)]} \mu'}{\varepsilon \vdash \mu \xrightarrow[\text{oper}]{p} \mu'}}$$

We shall see some examples of use of abstraction in the next section. We should emphasize the fact that in this rule p is a *program*, not just an atom call – compare with the rule for process calls. In other words there is no special synchronization discipline (e.g. mutual exclusion) prescribed for atom calls. This is because we want to achieve some synchronizations (rendez-vous) as a result of an abstraction; in order to operate an atom thus generally requires the cooperation of some other ones. Note that an atom call is a *recoverable* action: its operation holds only up to the completion of its code.

6. Communication Structures.

A communication structure generally consists of a data type, where the exchanged values are stored, and some procedures to send and receive through the structure. A typical example is that of a *cell* ("one slot buffer") which differs from the single shared variable in that one cannot write in an inhabited cell:

$$\begin{aligned}
& & \mathit{cell}(\tau) &= (\mathit{empty}\colon \mathsf{bool}{=}\mathit{tt} \text{ and } z\colon \tau) \\
&\text{with} & \mathit{put}(o\colon \mathit{cell}(\tau))(!x) &= (\mathsf{when}\ o.\mathit{empty}\ \mathsf{do}\ (o.z, o.\mathit{empty}){:=}(x, \mathit{ff})) \\
& & \mathit{get}(o\colon \mathit{cell}(\tau))(?y) &= (\mathsf{when}\ \neg o.\mathit{empty}\ \mathsf{do}\ (y, o.\mathit{empty}){:=}(o.z, \mathit{tt}))
\end{aligned}$$

The *empty* field acts as a semaphore, and thus *put* and *get* are mutually exclusive. This does not mean that a program cannot *perform* these atoms concurrently, e.g. in a compound action $(put(c)(e) \parallel x{:=}get(c))$: mutual exclusion only regards their operation.

A more instructive example is that of a CCS *port*. The specification of this communication structure is that there must be a rendez-vous between a sending and a receiving; a value is exchanged at that moment. A variant solution to that given in [8] is the following:

$$\begin{aligned} & port(\tau) = ((b{:}\,\mathsf{bool}{=}tt \ \mathsf{and}\ b'{:}\,\mathsf{bool}{=}tt)\ \mathsf{and}\ z{:}\,\tau) \\ \mathsf{with}\quad & send(o{:}\,port(\tau))(!x) = (\mathsf{when}\ o.b \wedge o.b'\ \mathsf{do}\ (o.z, o.b){:=}(x, f\!f))\ ;\ (\mathsf{when}\ \neg o.b'\ \mathsf{do}\ o.b{:=}tt) \\ & receive(o{:}\,port(\tau))(?y) = (\mathsf{when}\ (\neg o.b) \wedge o.b'\ \mathsf{do}\ (y, o.b'){:=}(o.z, f\!f))\ ;\ (\mathsf{when}\ o.b\ \mathsf{do}\ o.b'{:=}tt) \end{aligned}$$

Let p be a program which performs an action a which is either $send(u)(e)$, or $v{:=}receive(u)$, or the parallel product $(send(u)(e) \parallel v{:=}receive(u))$. We want to determine the behaviour of the configuration $(p\backslash\mu)$ – with $\mu(e) = \mathrm{v}$ –, in the context of an environment ε containing the definition above. This can only be obtained via the rule for closed configurations, hence we have to prove $\mu \xrightarrow[\mathsf{oper}]{a} \mu'$, by means of the abstraction rule. Let a' be the program we get from a by unfolding the calls to *send* or *receive* into their codes – respectively $s\ ;\ s'$ and $r\ ;\ r'$. Then a' operates on a memory where $u.b = tt$ and $u.b' = tt$; therefore if $a = (v{:=}receive(u))$ nothing can happen. If $a = send(u)(e)$ then the first instruction s of the *send* may operate, but not the second one; hence the configuration $(p\backslash\mu)$ cannot perform a since we cannot prove any operation of a on μ. If $a = (send(u)(e) \parallel v{:=}receive(u))$ then there is one – and only one – way to prove that a operates on μ, which is (omitting some irrelevant informations from the memory)

$$\left\{\begin{matrix} u.b = tt \\ u.b' = tt \\ u.z \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{s} \left\{\begin{matrix} u.b = f\!f \\ u.b' = tt \\ u.z = \mathrm{v} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{r} \left\{\begin{matrix} u.b = f\!f \\ u.b' = f\!f \\ u.z = \mathrm{v} \\ y = \mathrm{v} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{s'} \left\{\begin{matrix} u.b = tt \\ u.b' = f\!f \\ u.z = \mathrm{v} \\ y = \mathrm{v} \end{matrix}\right\} \xrightarrow[\mathsf{oper}]{r'} \left\{\begin{matrix} u.b = tt \\ u.b' = tt \\ u.z = \mathrm{v} \\ y = \mathrm{v} \end{matrix}\right\}$$

The interleaving is such that no other atom (*send* or *receive*) can interrupt this sequence. What allows to achieve this synchronization is the fact that atoms are "all-or-nothing" operations, and that no one can terminate without the cooperation of the other. There are some differences from the CCS port; the first one is that in an action an arbitrary number of pairs of complementary atoms may occur concurrently – this is because we use a SCCS-like notion of action. Another difference is that *receive* is a program, not a guard, and consequently the received value is assigned to a variable which may be a global one. Finally we do not assume that the co-occurrence of a *send* and a *receive* creates a new action – τ for CCS and 1 for SCCS/MEIJE.

Let us conclude this section with another example of communication scheme, that of *signal/wait*; the communication means is an *event* carrying a broadcast value. There are two possible interpretation: the *signal* action may be a blocking or a non-blocking one. According to the first interpretation no waiting process is bound to catch the signalled value (we do not assume any waiting discipline, queue or priority), whereas the second interpretation means that at least one waiting process must catch it. It is easy to formalize a "flashing" event, that is with a non-blocking *signal*:

$$\begin{aligned} & event(\tau) = (b{:}\,\mathsf{bool}{=}f\!f\ \mathsf{and}\ z{:}\,\tau) \\ \mathsf{with}\quad & signal(o{:}\,event(\tau))(!x) = (o.z{:=}x)\ ;\ (o.b{:=}tt)\ ;\ (o.b{:=}f\!f) \\ & wait(o{:}\,event(\tau))(?y) = (\mathsf{when}\ o.b\ \mathsf{do}\ y{:=}o.z) \end{aligned}$$

When the signalled value ought to be received the formalization is like that of a port:

$$\begin{aligned}
\mathit{event}'(\tau) &= ((b\colon \mathsf{bool}{=}\mathit{ff} \text{ and } b'\colon \mathsf{bool}{=}\mathit{tt}) \text{ and } z\colon\tau)\\
\text{with}\quad \mathit{signal}'(o\colon \mathit{event}'(\tau))(!x) &= (\text{when } \neg o.b \wedge o.b' \text{ do } (o.z, o.b){:=}(x, \mathit{tt}));\\
&\ (\text{when } \neg o.b' \text{ do } (o.b, o.b'){:=}(\mathit{ff}, \mathit{tt}))\\
\mathit{wait}'(o\colon \mathit{event}'(\tau))(?y) &= (\text{when } o.b \text{ do } (y, o.b'){:=}(o.z, \mathit{ff}))
\end{aligned}$$

In both cases the boolean b marks whether or not a value is available.

7. Conclusion.

This paper must be regarded as a preliminary work for many reasons. Certainly our proposal ought to be thoroughly studied, and enriched. For instance one would like to define a structure such as

```
$pstack(\tau) = ((empty\colon \mathsf{bool}{=}tt$ and $top\colon\tau)$ and $tail\colon pstack(\tau))$
inside
$fifo(\tau) = (left\colon pstack(\tau)$ and $right\colon pstack(\tau))$
    with   $ppush(s\colon pstack(\tau))(!x) = (s.empty, s.top, s.tail){:=}(ff, x, s)$
     and   $ppop(s\colon pstack(\tau)(?y) = (y, s){:=}(s.top, s.tail)$
     and   $siphon(f\colon fifo(\tau)) =$ while $\neg f.left.empty$ do
                   (let $z\colon\tau$ in $z{:=}ppop(f.left)$ ; $ppush(f.right)(z)$)
    inside $put(f\colon fifo(\tau))(!x) = ppush(f.left)(x)$
     and   $get(f\colon fifo(\tau)(?y) =$ when $\neg(f.left.empty \wedge f.right.empty)$ do
                   (if $f.right.empty$ then $siphon(f)$) ; $y{:=}ppop(f.right)$
```

The most conspicuous new keyword in this definition is inside; it introduces a group of definitions which are *private* to another group. In fact there is no special difficulty in dealing with the semantics of such a construct (see [13]). There would not be much more trouble in allowing the use of (while ... do ...) or (if ... then ...); we have already seen that since their termination depends on eval we should have to regard $\mathbb{1}$ as an action – which operates as an identity. Finally the code for the *siphon* atom above involves a (let ... in ...) subprogram. This construct introduces a *declaration of local identifiers*, such as $z\colon\tau$ – a declaration is nothing but a special kind of record type expression. A possible way to formalize the semantics of such a program is to pull up the local declarations to the root of the program – while renaming the identifiers with the same name but distinct scope. Then the behaviour of a configuration ((let δ in $p)\backslash\mu$) is that of $(p\backslash\mu[\delta])$: declarations are the syntactical means to build up a memory. This may have to be done dynamically, since local declarations may appear by unfolding a definition, as in the above example. It would be worthwhile to design a better formalization, for we have only proposed here a "centralized" point of view: in a configuration $(p\backslash\mu)$ there is just a global shared memory, and no local ones.

For a similar reason it would be interesting to add some analogue to the "ticking" primitive of MEIJE, since our proposal, while providing for mutual exclusion or inclusion of *operations*, does not allow to compel such constraints on *actions* – that is over the exec arrow. Equivalently we would have to add Milner's *synchronous product* $(p \times q)$: we have shown in [4] that this product, together with interleaving (or our "asynchronous" product $(p \,||\, q)$), sets up the two fundamental aspects of concurrency. While $(p \,||\, q)$ (*disjunctive* parallelism) relates to the notion of *shared* memory, the product $(p \times q)$ (*conjunctive* parallelism) rather corresponds to the idea of *distributed* memory.

As its stands our proposal lacks for a formalization of a static semantics, and especially of a type checking. For instance, let s be a variable of type semaphore (with $sem = (b\colon \mathsf{bool}{=}tt)$ with ...); then nothing prevents one from writing the program

$$(P(s) \,||\, s.b{:=}tt)$$

Needless to say, such an illegal manipulation of the "internal" structure of an object might be forbidden. In order to provide a true abstraction, one must ensure that a program only uses the abstract object by means of the suitable atoms (*cf.* [12]), and this is the rôle of the type checking mechanism.

Acknoledgments. I would like to thank Gérard Berry and Ilaria Castellani whose advice helped me to rewrite a previous version of this paper. I am greatly indebted to Georges Gonthier for pointing out some deficiencies of an earlier formalization.

REFERENCES

[1] D. AUSTRY, G. BOUDOL, *Algèbre de processus et synchronisation*, Theoret. Comput. Sci. 30 (1984) 91-131.

[2] G. BERRY, L. COSSERAT, *The* ESTEREL *Synchronous Programming Language and its Mathematical Semantics*, Seminar on Concurrency, Lecture Notes in Comput. Sci. 197 (1984) 389-448.

[3] G. BERRY, P.-L. CURIEN, *Theory and Practice of Sequential Algorithms: the Kernel of the Applicative Language* CDS, in Algebraic Methods in Semantics (M. Nivat & J. C. Reynolds, Eds), Cambridge University Press (1985) 35-87.

[4] G. BOUDOL, *Notes on Algebraic Calculi of Processes*, in Logics and Models of Concurrent Systems (K. Apt, Ed.) NATO ASI Series F13 (1985) 261-303, and INRIA Report 395.

[5] G. BOUDOL, *Calculs de processus et vérification*, Premier Colloque C^3 et Rapport INRIA 424 (1985).

[6] G. BOUDOL, I. CASTELLANI, *On the Semantics of Concurrency: Partial Orders and Transition Systems*, Rapport INRIA 550 (1986), to appear in Proc. CAAP 87.

[7] L. CARDELLI, P. WEGNER, *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys 17 (1985) 471-522.

[8] R.R. HOOGERWOORD, *An Implementation of Mutual Inclusion*, Inform. Proc. Letters 23 (1986) 77-80.

[9] P. JALOTTE, R.H. CAMPBELL, *Atomic Actions in Concurrent Systems*, 5th Intern. Conf. on Distributed Computing Systems (1985) 184-191.

[10] D.B. LOMET, *Process Structuring, Synchronization, and Recovery using Atomic Actions*, SIGPLAN Notices 12 (1977) 128-137.

[11] R. MILNER, *Calculi for Synchrony and Asynchrony*, Theoret. Comput. Sci. 25 (1983) 267-310.

[12] J.C. MITCHELL, G. PLOTKIN, *Abstract Types have Existential Type*, 12th POPL (1985) 37-51.

[13] G. PLOTKIN, *A Structural Approach to Operational Semantics*, Rep. Daimi FN-19, Aarhus University (1981).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique