Processes, Functions, and Datatypes*

Vasco T. Vasconcelos
Faculdade de Ciências da Universidade de Lisboa
Departamento de Informática
Portugal

Based on a name-passing calculus and on its typing system we show how to build several language constructors towards strongly-typed concurrent programming languages. In particular, we introduce a notion of datatype declaration, and show how to create values of a datatype and how to take such values apart. Further expressions include a form of abstraction and of application. Expressions evaluate to names and can replace any (non-binding) name in the calculus, thus achieving a clean incorporation of expressions in a name-passing calculus. A systematic translation of the derived constructors into the basic calculus provides for the semantics and for typing rules for the new constructors.

1. Introduction

Concurrent objects constitute a convenient tool to describe concurrent and distributed computations. Types enforce a discipline in the usage of the programming language constructors that not only provides for partial-correctness, but also helps in writing clear programs. Furthermore, a type for a program often gives some indication on the correct usage of the program.

The calculus of concurrent objects [28] aims at capturing fundamental notions present in concurrent objects. Programs are built from names and labels by means of a few constructors, namely: asynchronous labelled messages, objects composed of collections of labelled methods, concurrent composition, and an operator enabling to create a new name visible in some defined scope. The notions of process abstraction and variables over abstractions increase the flexibility of the calculus. They provide for (mutually) recursive processes and, for a form of (ML-like) let-definitions allowing to declare an abstraction once and to instantiate it several times in a given process.

The calculus of objects is implicitly typed in that no type information is present in programs but else inferred through a type system. Types, built from type-variables by means of a record constructor, describe the kind of messages objects may receive. While types are assigned to names, processes are assigned sets of name-type pairs (called typings) describing the types of the free names of the process. Abstractions (processes abstracted on a sequence of names) are assigned a sequence of types (one for each name in the sequence), constituting a monomorphic type for the abstraction. A notion of (ML-like) predicative polymorphism allows to abstract a monomorphic type on some type-variable, and to instantiate polymorphic types into different monomorphic types.

Typable programs are guaranteed not to suffer from runtime errors. These come into two forms: 'message not understood' (wrong label and/or wrong number of arguments in a message invocation), and 'instantiation not possible' (wrong number of arguments in a process instantiation).

Inspired on the π-calculus [13, 10], and on the actor model of computation [2, 9], TyCO (TYped Concurrent Objects) grows from the basic calculus of objects and its monomorphic type system [28], by introducing recursive types [5, 27], process abstractions, and predicative polymorphism [6, 24]. This constitutes what we call core-TyCO.

We then introduce a notion of expressions to be incorporated in the core calculus. Expressions simplify the writing of programs by hiding communication and synchronization details, thus achieving an higher-level programming standard. Expressions enable to construct and to take apart values built from datatype declarations (similar to those provided by functional programming languages such as ML [14], Miranda [23], and Haskell [11]), and include a form of application and of abstraction.

Expressions evaluate to names, and can be used wherever a non-binding name can. We develop a systematic method of translating expressions into core-TyCO, and, based on the translation and on the type system for core-TyCO, we present typing rules for the new syntactic category of expressions. Whenever possible we use a syntax similar to that of the ML pro-

*A preliminary version of this article appeared under the title "Typed Concurrent Objects".

gramming language [14]. This is particularly true for datatype declarations, and for case and abstraction expressions.

The outline of the paper is as follows. The next section introduces the calculus of objects, and section 3 its type system. Then section 4 describes datatype declarations; section 5 introduces expressions allowing to create values of a datatype and to take such values apart; section 6 incorporates expressions in the core calculus; and section 7 defines expression application and abstraction. Section 8 introduces convenient process constructors used in the encoding of expressions. The typing rules for all the new constructors are validated in section 9. Section 10 contains comparisons with related work, and the last section some pointers for future work.

2. The calculus of objects

This section presents a version of the calculus of objects [28], where variables over process abstractions provide for recursion and for a form of declaration [24]. The original formulation of the calculus uses replication to provide for unbounded computation power; the notion of abstractions (and variables over them) allows for mutually recursive definitions, a much more intuitive program constructor. By allowing to associate an abstraction to some variable (and to use the latter multiple times in a given process), declarations play a crucial role in the polymorphic type system (see section 3).

2.1. Syntax

Fix a denumerable set of *names*, a denumerable set of *variables over process abstractions*, and a set of *labels*. Possibly subscripted or primed, letters a, b, c and also u, v, w, x, y, z range over names. A sequence of names $v_1 \cdots v_n$ $(n \geq 0)$ is abbreviated to $\tilde{v}$; we assume that names are pairwise distinct in the sequence $\tilde{x}$. Letters X, Y range over abstraction variables, and letter l over labels. The set of *processes*, the set of *process abstractions*, the set of (collections of) *methods*, and the set of (collections of) *abstraction bindings* are given by the grammar

$$
\begin{array}{rcl}
P & ::= & a\,!\,l[\tilde{v}] \;\mid\; a?M \;\mid\; P \mid Q \;\mid\; \mathbf{0} \;\mid \\
 & & \textsf{new}\ x\ P \;\mid\; X[\tilde{v}] \;\mid\; A[\tilde{v}] \;\mid\; \textsf{def}\ D\ \textsf{in}\ P \\
A & ::= & (\tilde{x})P \\
M & ::= & \{l_1 = A_1, \ldots, l_n = A_n\} \\
D & ::= & X_1 = A_1\ \textsf{and} \cdots \textsf{and}\ X_n = A_n
\end{array}
$$

where $n \geq 0$, and the labels $l_1, \ldots, l_n$ and also the abstraction variables $X_1, \ldots, X_n$ are pairwise distinct.

Processes of the form $a\,!\,l[\tilde{v}]$ are called *messages*: name a is the *target*, and $l[\tilde{v}]$ is the *communication* of the message. Label l selects one particular method in the target object while the sequence of names $\tilde{v}$ constitutes the actual contents of the message. *Objects* are processes of the form $a?M$ where a represents the *location* of the object, and M a (finite, possibly empty) collection of *methods*. Each method, of the form $l = (\tilde{x})P$, is labelled by a different label l. The sequence of names $\tilde{x}$ represents the formal parameters, while process P represents the body of the method.

Objects as labelled sums and asynchronous labelled messages are the two distinctive constructors of the calculus. Intuitively, the result of the interaction of a message $a\,!\,l_i[\tilde{v}]$ and an object $a?\{l_1 = (\tilde{x}_1)P_1, \ldots, l_n = (\tilde{x}_n)P_n\}$ is the process P_i with names in $\tilde{x}_i$ (the formal parameters) replaced by those in $\tilde{v}$ (the actual parameters). Such an interaction is well defined only when l_i labels a method in the object, that is when $1 \leq i \leq n$, and when the lengths of $\tilde{v}$ and $\tilde{x}_i$ match.

The remaining process constructors are fairly standard in name-passing calculi. The *parallel composition* of P and Q, notation $P \mid Q$, denotes the processes P and Q running in parallel. A process of the form **new** x P is called *scope restriction*: it creates a new name x visible only in the scope determined by process P. *Inaction*, $\mathbf{0}$, denotes the terminated process.

A *declaration* of the form **def** $X_1 = A_1$ **and** $\cdots$ **and** $X_n = A_n$ **in** P binds abstractions A_i to variables X_i, allowing to define mutually recursive processes, and to instantiate multiple copies of these in process P.

Unlike processes, abstractions are not directly "executable", but else provide for templates from which processes can be instantiated. If A is an abstraction of the form $(\tilde{x})P$, then an *applied abstraction* $A[\tilde{v}]$ denotes the process P with names in $\tilde{x}$ replaced by those in $\tilde{v}$. Once again, such an instantiation is well defined only when $\tilde{v}$ and $\tilde{x}$ are sequences of the same length. Similarly, an *applied abstraction variable* $X[\tilde{v}]$ behaves as $A[\tilde{v}]$, if X is bound to A by some declaration.

Some abbreviations ease the task of program writing. A process **new** $x_1 \cdots$ **new** x_n P is abbreviated to **new** $x_1 \cdots x_n$ P, or to **new** $\tilde{x}$ P when n is not important. The particular label *val* is often omitted in messages — $a\,!\,[\tilde{v}]$ denotes the message $a\,!\,\textit{val}[\tilde{v}]$ — and in objects when it labels the object's only method — $a?\{A\}$ denotes the object $a?\{\textit{val} = A\}$. Parenthesis and square brackets enclosing the empty sequence of names are often omitted. As such, $a\,!\,l$ denotes the message $a\,!\,l[]$, X as a process denotes $X[]$, and P as an abstraction denotes $()P$. In examples, we use **def** $X = A$ as an abbreviation for **def** $X = A$ **in** P, where P denotes the "rest of the program."

Contrary to the conventional precedence of operators, we let the scope of a **new** extend as far to the right as possible. We feel this is more in line with variable declarations in conventional programming languages. Operator "|" binds tighter than **def** – **in**, so that **def** D **in** $P \mid Q$ means **def** D **in** $(P \mid Q)$.

The *one-place buffer cell* constitutes a simple example of a process. A cell is an object that holds a value and answers to two kinds of messages: write and read. Together with a write message comes the value to be stored in the cell. The read message carries a name intended to receive the value the cell holds, thus simulating a synchronous method call within an asynchronous calculus.

```
def Cell = (self value)
        self?{write= (newVal) Cell[self newVal],
              read= (replyTo) replyTo!val[value] |
                              Cell[self value]}
```

2.2. Semantics

Scope restriction and abstraction are the *name binding operators* in the calculus. A name x occurs *free* in a process if x is not in the scope of a (x) or a **new** x; otherwise x occurs *bound*. The *set of free names* in a process P, notation $\mathrm{fn}(P)$, is defined accordingly. We assume the usual notion of *simultaneous* substitution of names $\tilde{v}$ for the free occurrences of names $\tilde{x}$ in a process P, notation $P[\tilde{v}/\tilde{x}]$, defined only if the lengths of $\tilde{v}$ and $\tilde{x}$ match, and the names in $\tilde{x}$ are pairwise distinct.

Declarations constitute the only *abstraction-variable binding operator* in the calculus. A variable X occurs *free* in a process if X is not one of the X_i in the scope $A_1, \ldots, A_n, P$ of a process **def** $X_1 = A_1$ **and** $\cdots$ **and** $X_n = A_n$ **in** P; otherwise X occurs *bound*. The *set of free abstraction variables* in a process P, notation $\mathrm{fv}(P)$, is defined accordingly. *Alpha conversion*, both for names and for abstraction variables, is defined in the usual way.

Structural congruence simplifies the treatment of reduction. *Structural congruence* $\equiv$ is the smallest congruence relation over processes generated by the following rules.

1. $P \equiv Q$ if P is alpha-convertible to Q,
2. $P \mid Q \equiv Q \mid P$, $\quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$, $P \mid \mathbf{0} \equiv P$,
3. **new** x $\mathbf{0} \equiv \mathbf{0}$, $\quad$ **new** xy $P \equiv$ **new** yx P, (**new** x $P) \mid Q \equiv$ **new** x $P \mid Q$ if $x \notin \mathrm{fn}(Q)$,
4. **def** D **in** $\mathbf{0} \equiv \mathbf{0}$, $\quad$ **def** D **in new** x $P \equiv$ **new** x **def** D **in** P if $x \notin \mathrm{fn}(D)$, $\quad$ (**def** D **in** $P) \mid Q \equiv$ **def** D **in** $P \mid Q$ if $\mathrm{fv}(D) \cap \mathrm{fv}(Q) = \emptyset$,
5. $((\tilde{x})P)[\tilde{v}] \equiv P[\tilde{v}/\tilde{x}]$,
6. $(l_1 = A_1, l_2 = A_2) \equiv (l_2 = A_2, l_1 = A_1)$, $\quad (X_1 = A_1$ **and** $X_2 = A_2) \equiv (X_2 = A_2$ **and** $X_1 = A_1)$.

Rules 1 and 2 are conventional in process calculi [12]. The novelty is the rules for **def**; rule 4 implements for **def** what rule 3 does for **new** (cf. [7]). Rule 5 implements abstraction instantiation by replacing the formal parameters by the actuals.

One could be tempted to introduce a rule for joining definitions

$$\textbf{def } D \textbf{ in def } D' \textbf{ in } P \equiv \textbf{ def } D \textbf{ and } D' \textbf{ in } P$$

if $\mathrm{fv}(D) \cap \mathrm{fv}(D') = \emptyset$. Unfortunately polymorphic types for the variables bound in D may used in D' on the left but not on the right (see section 3, cf. [7]).

One-step reduction $\rightarrow$ is the smallest relation generated by the rules in figure 1. *Reduction* $\twoheadrightarrow$ is the relation $\equiv \cup \rightarrow^+$, where $\rightarrow^+$ is the transitive closure of $\rightarrow$.

The reception of messages by objects (followed by the selection of the appropriate method and the instantiation of the method's body) constitutes the basic communication mechanism of the calculus, as captured by axiom Com. Recursion is achieved via axiom Inst, allowing to replace occurrences of a variable X by an abstraction A, if X is bound to A in some definition. Message reception and instantiation can occur under parallel composition (rule Par), scope restriction (rule Res), and definitions (rule Def), but not in method bodies.

As an example, let A denote the cell abstraction defined above. We have the following reductions.

$$\textbf{def } \mathsf{Cell} = A \textbf{ in } \mathsf{Cell[s\ v]} \mid \mathsf{s!}\mathit{write}\mathsf{[u]} \twoheadrightarrow \textbf{def } \mathsf{Cell} = A \textbf{ in } \mathsf{Cell[s\ u]}$$

$$\textbf{def } \mathsf{Cell} = A \textbf{ in } \mathsf{Cell[s\ v]} \mid \mathsf{s!}\mathit{read}\mathsf{[r]} \twoheadrightarrow \textbf{def } \mathsf{Cell} = A \textbf{ in } \mathsf{Cell[s\ v]} \mid \mathsf{r!}\mathit{val}\mathsf{[v]}$$

2.3. Replication

Replication provides for a convenient means of defining persistent, stateless objects. Abbreviate a process

$$\textbf{def } X = a?\{l_1 = (\tilde{x}_1)P_1 \mid X, \ldots, l_n = (\tilde{x}_n)P_n \mid X\} \textbf{ in } X$$

where X does not occur free in $P_1, \ldots, P_n$ to

$$a?{*}\{l_1 = (\tilde{x}_1)P_1, \ldots, l_n = (\tilde{x}_n)P_n\}.$$

Using axioms Inst and Com, and rule Str we can easily see that

$$a\,!\,l_i[\tilde{v}] \mid a?{*}\{l_1 = A_1, \ldots, l_n = A_n\} \twoheadrightarrow A_i[\tilde{v}] \mid a?{*}\{l_1 = A_1, \ldots, l_n = A_n\},$$

which constitutes the rule for replication in the π-calculus [12].

3. Type assignment

This section presents two type systems for the calculus of objects. The *monomorphic type system* TA

$$\text{COM} \quad a\,!\,l_i[\tilde{v}] \mid a?\{l_1 = A_1, \ldots, l_n = A_n\} \to A_i[\tilde{v}] \qquad (1 \le i \le n)$$

$$\text{INST} \quad \textbf{def } D \textbf{ and } X = A \textbf{ in } X[\tilde{v}] \mid P \to \textbf{ def } D \textbf{ and } X = A \textbf{ in } A[\tilde{v}] \mid P$$

$$\text{PAR} \ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \text{RES} \ \frac{P \to P'}{\textbf{new } x\ P \to \textbf{new } x\ P'} \qquad \text{DEF} \ \frac{P \to P'}{\textbf{def } D \textbf{ in } P \to \textbf{def } D \textbf{ in } P'}$$

$$\text{STR} \ \frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}$$

FIG. 1. One-step reduction.

is a simple extension of the type system for the calculus without abstractions [28]. The *predicative polymorphic type system* $\text{TA}_\forall$ is inspired on the Damas-Milner type system for the ML programming language [6], and on the predicative polymorphic type system for the π-calculus [24].

3.1. Monomorphic type assignment

Types for names are built from an infinite set of *type-variables*, and the set of labels introduced in the previous section, by means of a record constructor and a recursive type constructor.

Fix a denumerable set of *type variables*. Letter t ranges over type variables, and letters α, β over types. A sequence of types $\alpha_1 \cdots \alpha_n$ ($n \ge 0$) is abbreviated to $\tilde{\alpha}$. The *set of types for names* is given by the grammar

$$\alpha \quad ::= \quad t \quad | \quad \{l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n\} \quad | \quad \mu t.\alpha$$

where $n \ge 0$, and the labels $l_1, \ldots, l_n$ are pairwise distinct.

A type of the form $\{l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n\}$ is intended to describe names representing locations of objects containing n methods labelled with $l_1, \ldots, l_n$, and whose arguments of method l_i belong to the types $\tilde{\alpha}_i$. When $\tilde{\alpha}$ is the empty sequence of types, a record component of the form $l : \tilde{\alpha}$ is abbreviated to l. When n and the record components $l_2 : \tilde{\alpha}_2, \ldots, l_n : \tilde{\alpha}_n$ are not important, we abbreviate a type $\{l_1 : \tilde{\alpha}_1, l_2 : \tilde{\alpha}_2, \ldots, l_n : \tilde{\alpha}_n\}$ to $\{l_1 : \tilde{\alpha}_1, \ldots\}$.

Types are interpreted as rational (or regular infinite) trees. A type of the form $\mu t.\alpha$ (with $\alpha \ne t$) denotes the rational tree solution of the equation $t = \alpha$ [5, 27]. If α is a type, denote by α^* its associated rational tree. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on types, by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$.

Type assignments to names are formulae $x \colon \alpha$, for x a name and α a type. *Typings*, denoted by Γ, Δ, are sets of type assignments where no name occurs twice. If $\tilde{x} = x_1 \cdots x_n$ is a sequence of pairwise distinct names, and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ is a sequence of types, we write $\{\tilde{x} \colon \tilde{\alpha}\}$ for the typing $\{x_1 \colon \alpha_1, \ldots, x_n \colon \alpha_n\}$, and $\Gamma \cdot \tilde{x} \colon \tilde{\alpha}$ for the typing $\Gamma \cup \{\tilde{x} \colon \tilde{\alpha}\}$, provided that names in $\tilde{x}$ do not appear in Γ. The typing $\Gamma \setminus x$ is obtained from Γ by removing the type assignment associated with x. The expression $\Gamma(a)$ denotes the type α if $a : \alpha \in \Gamma$, and $\Gamma(a_1 \cdots a_n)$ represents the sequence of types $\alpha_1 \cdots \alpha_n$ if $a_i : \alpha_i \in \Gamma$, for $i = 1 \ldots n$.

Type assignments to abstraction variables are formulae $X \colon \tilde{\alpha}$, for X an abstraction variable and $\tilde{\alpha}$ a sequence of types. *Bases*, denoted by B, are sets of type assignments to variables where no variable occurs twice. Similarly to typings, we write $B \cdot X \colon \tilde{\alpha}$ for $B \cup \{X \colon \tilde{\alpha}\}$, provided that X does not appear in B, write $B \setminus X$ for the basis B with no occurrences of X, and write $B(X)$ for $\tilde{\alpha}$ if $X : \tilde{\alpha} \in B$.

Typing assignments to processes are statements $B, \Gamma \vdash P$, whereas *type assignment to abstractions* (or *methods*) are statements $B, \Gamma \vdash A : \tilde{\alpha}$ (or $B, \Gamma \vdash M : \tilde{\alpha}$). The *monomorphic type system* TA is composed of the axioms and rules in figure 2, where, in the APP-rule, U stands for an abstraction or an abstraction variable.

Denoting by A the cell abstraction in the example of section 2, a simple deduction ending with the ABS-rule shows that

$$\begin{aligned} &\mathsf{Cell} : (\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t), \emptyset \vdash \\ &\qquad\qquad A : \{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t. \end{aligned}$$

The type for the cell abstraction conveys the idea that cells are objects holding values of type t, and composed of two methods *write* and *read*. Method *write* accepts names of type t, and method *read* accepts names capable of receiving *val*-labelled messages carrying names of type t. This is not the only possible type for the cell abstraction: we may not only replace type variable t by an arbitrary type α, but also extend the type of the

$$\textsc{Msg} \quad B, \Gamma \vdash a\,!\,l_i[\tilde{v}] \quad (\Gamma(a) \approx \{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\}, \Gamma(\tilde{v}) \approx \tilde{\alpha}_i) \qquad \textsc{Nil} \quad B, \Gamma \vdash \mathbf{0}$$

$$\textsc{Obj} \quad \frac{B, \Gamma \vdash M : \alpha}{B, \Gamma \vdash a?M} \; (\Gamma(a) \approx \alpha) \qquad \textsc{Res} \quad \frac{B, \Gamma \vdash P}{B, \Gamma \setminus x \vdash \textsf{\textbf{new}}\ x\ P} \qquad \textsc{Par} \quad \frac{B, \Gamma \vdash P \qquad B, \Gamma \vdash Q}{B, \Gamma \vdash P \mid Q}$$

$$\textsc{Def} \quad \frac{B, \Gamma \vdash A_1 : \tilde{\alpha}_1 \quad \cdots \quad B, \Gamma \vdash A_n : \tilde{\alpha}_n \quad B, \Gamma \vdash P}{B \setminus \tilde{X}, \Gamma \vdash \textsf{\textbf{def}}\ X_1 = A_1\ \textsf{\textbf{and}} \cdots \textsf{\textbf{and}}\ X_n = A_n\ \textsf{\textbf{in}}\ P} \; (B(X_i) \approx \tilde{\alpha}_i)$$

$$\textsc{Abs} \quad \frac{B, \Gamma \cdot \tilde{x} : \tilde{\alpha} \vdash P}{B, \Gamma \vdash (\tilde{x})P : \tilde{\alpha}} \qquad \textsc{Var} \quad B \cdot X \colon \tilde{\alpha}, \Gamma \vdash X : \tilde{\alpha} \qquad \textsc{App} \quad \frac{B, \Gamma \vdash U : \tilde{\alpha}}{B, \Gamma \vdash U[\tilde{v}]} \; (\Gamma(\tilde{v}) \approx \tilde{\alpha})$$

$$\textsc{Meth} \quad \frac{B, \Gamma \vdash A_1 : \tilde{\alpha}_1 \quad \cdots \quad B, \Gamma \vdash A_n : \tilde{\alpha}_n}{B, \Gamma \vdash \{l_1 = A_1, \ldots, l_n = A_n\} : \{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\}}$$

FIG. 2. Monomorphic type system.

read method to obtain, say[1]

$$\begin{aligned}&\mathsf{Cell} : (\{\mathit{write} : \alpha, \mathit{read} : \{\mathit{val} : \alpha, \mathit{foo} : \beta\}\}\, \alpha), \emptyset \vdash \\ &\qquad A : \{\mathit{write} : \alpha, \mathit{read} : \{\mathit{val} : \alpha, \mathit{foo} : \beta\}\}\, \alpha.\end{aligned}$$

3.2. Replication

In the previous section we introduced replication as a derived constructor. We now present an admissible rule for the new constructor that constitutes the usual rule for replication in the π-calculus [27].

$$\textsc{Rep} \quad \frac{B, \Gamma \vdash a?M}{B, \Gamma \vdash a?{*}M}$$

To see why Rep is admissible notice that the final part of a deduction for the numerator $B, \Gamma \vdash a?M$ must be of the form

$$\dfrac{\dfrac{\dfrac{B, \Gamma \cdot \tilde{x}_1 : \tilde{\alpha}_1 \vdash P_1}{B, \Gamma \vdash (\tilde{x}_1)P_1 : \tilde{\alpha}_1} \quad \cdots \quad \dfrac{B, \Gamma \cdot \tilde{x}_n : \tilde{\alpha}_n \vdash P_n}{B, \Gamma \vdash (\tilde{x}_n)P_n : \tilde{\alpha}_1}}{B, \Gamma \vdash \{l_1 = A_1, \ldots, l_n = A_n\} : \{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\}}}{B, \Gamma \vdash a?\{l_1 = A_1, \ldots, l_n = A_n\}}$$

where $\Gamma(a) \approx \{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\}$. Then, starting from $B \cdot X : \epsilon, \Gamma \cdot \tilde{x}_i : \tilde{\alpha}_i \vdash P_i$ (obtained from the hypotheses via the typing-lemma 1 at the end of this section) and axiom Var, by using rules Par, Abs, Meth, Obj, Abs, and Def, one gets a deduction for the denominator $B, \Gamma \vdash a?{*}M$.

3.3. Polymorphic type assignment

A simple extension of the monomorphic type system allows to abstract on a particular type-variable a type for an abstraction, and to instantiate an abstracted type into different monomorphic types. Types for abstractions now fall into two classes: *monomorphic types* as sequences of types for names, and *polymorphic types* constructed using $\forall$.

The set of *polymorphic types for abstractions* is given by the following grammar.

$$\sigma \quad ::= \quad \tilde{\alpha} \quad | \quad \forall t.\sigma$$

As a consequence of the definition, universal quantifiers can only occur at the top level of types: types for abstractions are of the form $\forall t_1. \cdots .\forall t_n.\tilde{\alpha}$ for $n \geq 0$.

A type variable t occurs *free* in a type if t is not in the scope of a μt or a $\forall t$; otherwise t occurs *bound*. A type-variable occurs free in a typing Γ (respectively basis B) if it occurs free in some type in Γ (respectively B). We assume the usual notion of substitution of a type α for the free occurrences of a type-variable t in a type σ, and denote it by $\sigma[\alpha/t]$.

The *polymorphic type system* $\mathrm{TA}_\forall$ is defined by the axioms and rules in system TA with the monomorphic type in rule Var (but not in Msg, Meth, Obj, Abs, or App) replaced by a polymorphic type, by allowing polymorphic types in bases (but not in typings), by replacing rule Def by rule $\forall$-Def, and by adding rules $\forall$-intro and $\forall$-elim in figure 3.

One could be tempted to define $\mathrm{TA}_\forall$ by simply allowing polymorphic types in the Def-rule, but then we would run against the classical problem of polymorphic recursion variables [16].

Regaining the example of the buffer cell where we left it, we may use rule $\forall$-intro to abstract the type of the value the cell holds, to obtain,

$$\begin{aligned}&\mathsf{Cell} : (\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t), \Gamma \vdash A : \\ &\qquad\qquad \forall t.(\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t).\end{aligned}$$

Then, using twice the $\forall$-elim rule we have

$$\Gamma \vdash \textsf{\textbf{def}}\ \mathsf{Cell} = A\ \textsf{\textbf{in}}\ \mathsf{Cell[n\ 1]} \mid \mathsf{Cell[b\ True]}$$

$$\forall\text{-INTRO}\quad \frac{B, \Gamma \vdash A : \sigma}{B, \Gamma \vdash A : \forall t.\sigma}\ (t \text{ not free in } B, \Gamma) \qquad\qquad \forall\text{-ELIM}\quad \frac{B, \Gamma \vdash X : \forall t.\sigma}{B, \Gamma \vdash X : \sigma[\alpha/t]}$$

$$\forall\text{-DEF}\quad \frac{B, \Gamma \vdash A_1 : \sigma_1 \qquad \cdots \qquad B, \Gamma \vdash A_n : \sigma_n \qquad B \setminus \tilde{X} \cdot \tilde{X} : \tilde{\sigma}, \Gamma \vdash P}{B \setminus \tilde{X}, \Gamma \vdash \textsf{def}\ X_1 = A_1\ \textsf{and} \cdots \textsf{and}\ X_n = A_n\ \textsf{in}\ P}\ \ (B(X_i) \approx \tilde{\alpha}_i; \sigma_i = \forall \tilde{t}_i.\tilde{\alpha}_i)$$

FIG. 3. New rules for the polymorphic type system.

where Γ is the typing $\{\mathsf{n} : \{\mathit{write} : \mathsf{nat}, \mathit{read} : \{\mathit{val} : \mathsf{nat}\}\}, \mathsf{1} : \mathsf{nat}, \mathsf{b} : \{\mathit{write} : \mathsf{bool}, \mathit{read} : \{\mathit{val} : \mathsf{bool}\}\}, \mathsf{True} : \mathsf{bool}\}$, for some types nat and bool.

3.4. Some properties of the type systems

We conclude this section with a brief overview of some important properties of the type systems. The proofs for the monomorphic system are to be found in references [25, 28]. The proofs for the polymorphic system (in the context of the π-calculus) appear in reference [24].

Lemma 1 Typing Lemma. *Let $B, \Gamma \vdash P$. Then*

1. $B, \Gamma \cdot x : \alpha \vdash P$ *and* $B \cdot X : \sigma, \Gamma \vdash P$*;*
2. *If* $x \notin \mathrm{fn}(P)$*, then* $B, \Gamma \setminus x \vdash P$*;*
3. *If* $X \notin \mathrm{fv}(P)$*, then* $B \setminus X, \Gamma \vdash P$*.*

Lemma 2 Substitution Lemma. *If $B, \Gamma \cdot y : \alpha \vdash P$ and $\Gamma(x) = \alpha$, then $B, \Gamma \vdash P[x/y]$.*

Subject-reduction ensures that a typing for a process does not change as the process is reduced.

Theorem 3 Subject-reduction. *If $B, \Gamma \vdash P$ and $P \twoheadrightarrow Q$, then $B, \Gamma \vdash Q$.*

As a corollary, typable processes do not encounter type errors at runtime. We say a process P contains a possible *runtime error* if

$$P \twoheadrightarrow \textsf{def}\ D_1\ \textsf{in} \cdots \textsf{def}\ D_n\ \textsf{in new}\ \tilde{y}\ a?M \mid a\,!\,l[\tilde{v}] \mid Q$$

and l labels no method in M, or

$$P \twoheadrightarrow \textsf{def}\ D_1\ \textsf{in} \cdots \textsf{def}\ D_n\ \textsf{in new}\ \tilde{y}\ ((\tilde{x})P)[\tilde{v}] \mid Q$$

and the lengths of $\tilde{x}$ and $\tilde{v}$ do not match.

Corollary 4. *If $\Gamma \vdash P$, then P contains no runtime error.*

Decidability of typing assignment (given B, P, and Γ, is it the case that $B, \Gamma \vdash P$?) and *computability of typing existence* (given B and P, is there a Γ such that $B, \Gamma \vdash P$?) are two properties verified by system $\mathrm{TA}_\forall$ despite the fact that the system possess no simple notion of subsumption (and hence of principal typings). Such notions can be obtained by introducing, for example, constraints on the types that may replace a given type-variable, in the form of Ohori's *kinds* [17, 28] (cf. endnote 11).

4. Datatype declaration

This section introduces a notion of ML-like datatype declarations, from which values can be constructed.

Fix a set of *type identifiers*, and a set of *type constructors*, subset of the set of labels. Letter T ranges over type identifiers, and letter C over type constructors. Add to processes *datatype declarations* as follows,

$$P \quad ::= \quad \cdots \quad | \quad \textsf{datatype}\ \mathrm{TB}_1\ \textsf{and} \cdots \textsf{and}\ \mathrm{TB}_n\ \textsf{in}\ P$$

where *type bindings* TB are of the form

$$T = \{C_1 : \tilde{T}_1, \ldots, C_n : \tilde{T}_n\}$$

and the constructors $C_1, \ldots, C_n$ are pairwise distinct. Similarly to types, we abbreviate $C : \tilde{T}$ to C when $\tilde{T}$ is the empty sequence. Notice that datatype declarations allow the declaration of mutually recursive types, by providing for the simultaneous definition of several type bindings.

Datatype declarations are readily translated into core-TyCO processes. In the sequel we use the mapping $\{\!|\cdot|\!\}$ to translate new process constructors into the base calculus.

$$\{\!|C : \tilde{T}|\!\} \stackrel{\text{def}}{=} C = (v\tilde{x})v?{*}\{(r)r\,!\,C[\tilde{x}]\}$$

$$\{\!|T = \{C_1 : \tilde{T}_1, \ldots, C_n : \tilde{T}_n\}|\!\} \stackrel{\text{def}}{=} \{\!|C_1 : \tilde{T}_1|\!\}\ \textsf{and} \ldots \textsf{and}\ \{\!|C_n : \tilde{T}_n|\!\}$$

$$\{\!|\textsf{datatype}\ \mathrm{TB}_1\ \textsf{and} \cdots \textsf{and}\ \mathrm{TB}_n\ \textsf{in}\ P|\!\} \stackrel{\text{def}}{=} \textsf{def}\ \{\!|\mathrm{TB}_1|\!\}\ \textsf{and} \cdots \textsf{and}\ \{\!|\mathrm{TB}_n|\!\}\ \textsf{in}\ \{\!|P|\!\}$$

In the translation of type bindings, $\tilde{x}$ and $\tilde{T}$ are sequences of the same length; the new typing rules below ensure that the names in $\tilde{x}$ are of types in $\tilde{T}$. We take the convention that all new names introduced by the encodings are fresh. A type constructor $C : \tilde{T}$ is encoded as a replicated object replying $C[\tilde{x}]$ to invocations; replication provides for multiple evaluations of the datatype. Notice that we use the actual constructor C to label the message resulting from the evaluation of a type constructor.

A datatype for boolean values may be defined as

datatype Bool = {True, False}

to be encoded as a pair of abstraction bindings, one for the True constructor, the other for the False constructor.

def True = (b) b?*{(r) r ! *True*}
and False = (b) b?*{(r) r ! *False*}

An object instance of True, when invoked, replies *True* stating "I am the value true," and similarly for instances of False. Notice that False is an abstraction variable whereas *False* is a label.

A datatype for lists of boolean values can be defined as

datatype BoolList = {Nil, Cons: Bool BoolList}

and translated into the following pair of bindings.

def Nil = (l) l?*{(r) r ! *Nil*}
and Cons = (l head tail) l?*{(r) r ! *Cons*:[head tail]}

Natural number can be defined by the declaration **datatype** Nat = {Zero, Succ: Nat}.

Types for datatype declarations can be extracted from the new syntax. To keep track of type bindings we need a new environment. *Type assignments to type identifiers* are formulae $T : \alpha$. *Type bindings*, denoted by D, are sets of type assignments to type identifiers where no type identifier occurs twice. The *type assignment system for datatype declarations* is composed of the axiom and the rule in figure 4.

Looking at the encoding of the Bool datatype, one can easily see that boolean values are objects capable of receiving *val*-labelled messages containing names capable of receiving *True*- and *False*-labelled messages, as captured by the type

$$\mathsf{bool} \stackrel{\text{def}}{=} \{\mathit{val} : \{\mathit{True}, \mathit{False}\}\} \approx \mu t.\{\mathit{val} : \{\mathit{True}, \mathit{False}\}\}$$

in the below instance of the TYBIND-axiom,

$$D_0, B_0 \vdash \mathsf{Bool} \ = \{\mathsf{True}, \mathsf{False}\}$$

where the initial bindings for type identifiers D_0 is $\{\mathsf{Bool} : \mathsf{bool}\}$, and the initial basis B_0 is $\{\mathsf{True} : \mathsf{bool}, \mathsf{False} : \mathsf{bool}\}$. Notice that bool is an abbreviation for a type expression, whereas Bool is a type identifier.

Similarly, lists are objects capable of receiving *val*-labelled messages, replying either *Nil* or *Cons*: head tail. The BoolList declaration dictates that head should be an element of the list, and that tail should itself be a list, yielding the type

$$\mathsf{boolList} \stackrel{\text{def}}{=} \mu t.\{\mathit{val} : \{\mathit{Nil}, \mathit{Cons} : \mathsf{bool}\ t\}\}$$

in the following instance of the TYBIND-axiom

$$D_0, B_0 \vdash \mathsf{BoolList} \ = \{\mathsf{Nil}, \mathsf{Cons} : \mathsf{Bool}\ \mathsf{BoolList}\}$$

where we have added $\mathsf{BoolList} : \mathsf{boolList}$ to D_0, and $\mathsf{Nil} : \mathsf{boolList}, \mathsf{Cons} : \mathsf{boolList}$ to B_0.

5. Expressions I - Creating and taking apart values of a datatype

This section presents expressions that allow to create values of a datatype and to take such values apart. Section 7 completes the presentation of expressions.

The set of *expressions* is given by the following grammar, where letter E ranges over expressions.

$$\begin{aligned} E \ ::= \ & a \mid C[\tilde{E}] \mid \\ & \textbf{case}\ E\ \textbf{of}\ \{C_1 \Rightarrow (\tilde{x}_1)E_1, \ldots, C_n \Rightarrow (\tilde{x}_n)E_n\} \mid \\ & \ldots \end{aligned}$$

Names constitute the basic expressions. If C is a type constructor defined by some datatype declaration, and $\tilde{E}$ is a sequence of expressions, then $C[\tilde{E}]$ is a *constructed value*. The case expression is used to take apart a value of a datatype, choosing a particular branch E_i according to the value of E.

Intuitively, an expression is evaluated to a name in some context where the expression is supposed to be used. In the sequel, the map $[\![E]\!]_v P$ is used to translate an expression E, a name v, and a process P into a base process. Notice that $\{\!|\cdot|\!\}$ translates new process constructors, whereas $[\![\cdot]\!]$ translates expressions.

$$\begin{aligned} [\![a]\!]_v P &\stackrel{\text{def}}{=} P[a/v] \\ [\![C[\tilde{E}]]\!]_v P &\stackrel{\text{def}}{=} \textbf{new}\ v\ \{\!|v := C[\tilde{E}]|\!\} \mid P \\ [\![E_1 \cdots E_n]\!]_{v_1 \cdots v_n} P &\stackrel{\text{def}}{=} [\![E_1]\!]_{v_1}(\cdots([\![E_n]\!]_{v_n} P)\cdots) \end{aligned}$$

$$[\![\textbf{case}\ E\ \textbf{of}\ \{C_1 \Rightarrow (\tilde{x}_1)E_1, \ldots, C_n \Rightarrow (\tilde{x}_n)E_n\}]\!]_v P \stackrel{\text{def}}{=}$$
$$\textbf{branch}\ E\ \textbf{into}\ \{C_1 = (\tilde{x}_1)[\![E_1]\!]_v P, \ldots, C_n = (\tilde{x}_n)[\![E_n]\!]_v P\}$$
$$\{\!|\textbf{branch}\ E\ \textbf{into}\ M|\!\} \stackrel{\text{def}}{=} \textbf{new}\ r\ [\![E]\!]_u(u\,!\,[r]) \mid r?M$$
$$\{\!|v := C[\tilde{E}]|\!\} \stackrel{\text{def}}{=} [\![\tilde{E}]\!]_{\tilde{v}}(C[v\tilde{v}])$$

A constructed value $C[\tilde{E}]$ is encoded into a name v in a context P as the parallel composition of an *environment entry* $v := C[\tilde{E}]$ and the process P itself, under a **new** v. Environment entries (cf. [12]) assign values to names that may be shared; such values must then be replicated. An environment entry of the above form evaluates expressions $\tilde{E}$ into names $\tilde{v}$ and instantiates a copy of the abstraction C located at v with actual parameters $\tilde{v}$. For example, a list Cons[a Cons[b Nil]] composed of elements a and b, translates into an object located at u in a context P as follows.

new w Nil[w] | **new** v Cons[v b w] |
new u Cons[u a v] | P

$$\text{TyBind}\quad D\cdot T:\alpha, B\cdot C_1:\alpha\tilde\beta_1\cdots C_n:\alpha\tilde\beta_n \vdash T=\{C_1:\tilde T_1,\dots,C_n:\tilde T_n\}\quad \begin{array}{l}(\tilde\beta_i=(D\cdot T:t)(\tilde T_i);\ t\text{ fresh}\\ \alpha\approx\mu t.\{\mathit{val}:\{C_1:\tilde\beta_1,\dots,C_n:\tilde\beta_n\}\})\end{array}$$

$$\text{DType}\quad \frac{D,B\vdash \mathrm{TB}_1\quad\cdots\quad D,B\vdash\mathrm{TB}_n\quad D,B,\Gamma\vdash P}{D\setminus\tilde T,B,\Gamma\vdash \textbf{datatype}\ \mathrm{TB}_1\ \textbf{and}\cdots\textbf{and}\ \mathrm{TB}_n\ \textbf{in}\ P}\quad(\tilde T\text{ bound in }\mathrm{TB}_1\dots\mathrm{TB}_n)$$

FIG. 4. Type assignment to datatype declarations.

A case expression is encoded into a *branch process*. Such a process evaluates E into name u, and then branches according to this. The branching is achieved by forming a message $u\,!\,[r]$; the u-located object eventually replies $r\,!\,C_i[\tilde v]$, thus selecting one method in the newly created r-located object. The result of the case-expression is the result of evaluating E_i with $\tilde x_i$ replaced by $\tilde v$. For example, a case-expression testing whether a given list l is empty[2]

case l **of** $\{$Nil$\Rightarrow$ True, Cons$\Rightarrow$ (_ _) False$\}$

is translated into name v in the context of process P as

new r l![r] | r?$\{$*Nil*= **new** v True[v] | P,
Cons= (_ _) **new** v False[v] | $P\}$.

The encoding into core-TyCO not only provides for the semantics of expressions, but also allows to derive type rules for the new expressions. Since expressions stand for names and we assign types to names, we should as well assign types to expressions. Therefore, *typing assignments to expressions* are statements $B,\Gamma\vdash E:\alpha$. The *type system for expressions* $\mathrm{TA}_{\mathrm{exp}}$ is composed of the axiom and rules in figure 5 (rules ExpAbs, ExpApp, and Rec are introduced in section 7).

A simple deduction using only the Name-axiom and the Con-rule shows that

$$B_0,\mathsf{a}:\mathsf{bool}\cdot\mathsf{b}:\mathsf{bool}\vdash \mathsf{Cons[a\ Cons[b\ Nil]]}:\mathsf{boolList}.$$

Similarly, using the Con-rule (in its axiom form) and the Case-rule, we have that

B_0, l : boolList $\vdash$
case l **of** $\{$Nil$\Rightarrow$ True, Cons$\Rightarrow$ (_ _) False$\}$: bool

since $\{\mathit{val}:\{\mathit{Nil},\mathit{Cons}:\mathsf{bool\ boolList}\}\}\approx\mathsf{boolList}$.

Conditional expressions are just particular instances of case-expressions. An expression of the form

if E **then** E_1 **else** E_2

is simply an abbreviation for

case E **of** $\{\mathit{True}\Rightarrow E_1,\mathit{False}\Rightarrow E_2\}$.

The expected type rule for conditionals is a particular form of the Case-rule.

$$\text{Cond}\quad\frac{B,\Gamma\vdash E:\mathsf{bool}\quad B,\Gamma\vdash E_1:\alpha\quad B,\Gamma\vdash E_2:\alpha}{B,\Gamma\vdash(\textbf{if}\ E\ \textbf{then}\ E_1\ \textbf{else}\ E_2):\alpha}$$

6. Incorporating expressions in the base calculus

Wherever in the base calculus we have a name in a *non-binding position* we may now have an expression in the calculus with expressions. In particular, we allow expressions to appear at the location of objects $E?M$, as arguments to instantiate abstractions $X[\tilde E]$ or $A[\tilde E]$, and at the target and contents of messages $E\,!\,l[\tilde E]$.

The set of *processes with expressions* is obtained from the set of processes by replacing objects $a?M$, messages $a\,!\,l[\tilde v]$, and abstraction instantiation $X[\tilde v], A[\tilde v]$ by their new forms

$$E?M,\quad E\,!\,l[\tilde E],\quad X[\tilde E],\quad\text{and}\quad A[\tilde E].$$

We use P,Q to range over processes with expressions, relying on the context to distinguish whether we are referring to base-processes or processes with expressions.

Processes with expressions can be readily translated into the base calculus. The translation $\{\!|\cdot|\!\}$ from processes with expressions into plain processes is inductively defined as follows.

$$\begin{aligned}
\{\!|E?M|\!\} &\stackrel{\text{def}}{=} [\![E]\!]_v(v?\{\!|M|\!\})\\
\{\!|E\,!\,l[\tilde E]|\!\} &\stackrel{\text{def}}{=} [\![E\tilde E]\!]_{v\tilde v}(v\,!\,l[\tilde v])\\
\{\!|X[\tilde E]|\!\} &\stackrel{\text{def}}{=} [\![\tilde E]\!]_{\tilde v}(X[\tilde v])\\
\{\!|A[\tilde E]|\!\} &\stackrel{\text{def}}{=} [\![\tilde E]\!]_{\tilde v}(\{\!|A|\!\}[\tilde v])\\
\{\!|\textbf{new}\ x\ P|\!\} &\stackrel{\text{def}}{=} \textbf{new}\ x\ \{\!|P|\!\}\\
\{\!|P\mid Q|\!\} &\stackrel{\text{def}}{=} \{\!|P|\!\}\mid\{\!|Q|\!\}\\
\{\!|\mathbf{0}|\!\} &\stackrel{\text{def}}{=} \mathbf{0}\\
\{\!|\textbf{def}\ D\ \textbf{in}\ P|\!\} &\stackrel{\text{def}}{=} \textbf{def}\ \{\!|D|\!\}\ \textbf{in}\ \{\!|P|\!\}\\
\{\!|(\tilde x)P|\!\} &\stackrel{\text{def}}{=} (\tilde x)\{\!|P|\!\}\\
\{\!|\{l_1=A_1,\dots,l_n=A_n\}|\!\} &\stackrel{\text{def}}{=} \{l_1:\{\!|A_1|\!\},\dots,l_n:\{\!|A_n|\!\}\}
\end{aligned}$$

$$\text{NAME} \quad B,\Gamma \vdash a : \alpha \quad (\Gamma(a) \approx \alpha) \qquad\qquad \text{CON} \quad \frac{B,\Gamma \vdash \tilde{E} : \tilde{\alpha}}{B,\Gamma \vdash C[\tilde{E}] : \alpha} \quad (B(C) \approx \alpha\tilde{\alpha})$$

$$\text{CASE} \quad \frac{B,\Gamma \vdash E : \{\mathit{val} : \{C_1 : \tilde{\alpha}_1, \ldots, C_n : \tilde{\alpha}_n\}, \ldots\} \qquad B,\Gamma \cdot \tilde{x}_1 : \tilde{\alpha}_1 \vdash E_1 : \beta \qquad \cdots \qquad B,\Gamma \cdot \tilde{x}_n : \tilde{\alpha}_n \vdash E_n : \beta}{B,\Gamma \vdash (\textbf{case } E \textbf{ of } \{C_1 \Rightarrow (\tilde{x}_1)E_1, \ldots, C_n \Rightarrow (\tilde{x}_n)E_n\}) : \beta}$$

$$\text{EXPSEQ} \quad \frac{B,\Gamma \vdash E_1 : \alpha_1 \qquad \cdots \qquad B,\Gamma \vdash E_n : \alpha_n}{B,\Gamma \vdash E_1 \cdots E_n : \alpha_1 \cdots \alpha_n} \qquad\qquad \text{REC} \quad \frac{B,\Gamma \cdot x : \alpha \vdash E : \alpha}{B,\Gamma \vdash \textbf{rec } x.E : \alpha}$$

$$\text{EXPABS} \quad \frac{B,\Gamma \cdot \tilde{x} : \tilde{\alpha} \vdash \tilde{E} : \tilde{\beta}}{B,\Gamma \vdash (\textbf{fn } \tilde{x} \Rightarrow l[\tilde{E}]) : \{\mathit{val} : \tilde{\alpha}\{l : \tilde{\beta}, \ldots\}\}} \qquad \text{EXPAPP} \quad \frac{B,\Gamma \vdash E : \{l : \tilde{\alpha}\{\mathit{val} : \beta\}, \ldots\} \qquad B,\Gamma \vdash \tilde{E} : \tilde{\alpha}}{B,\Gamma \vdash (E\,!\,l[\tilde{E}]) : \beta}$$

FIG. 5. Type system for expressions.

$$\{\!|X_1 = A_1 \textbf{ and} \cdots \textbf{and } X_n = A_n|\!\} \stackrel{\text{def}}{=} X_1 = \{\!|A_1|\!\} \textbf{ and} \cdots \textbf{and } X_n = \{\!|A_n|\!\}$$

For example, the message

map ! [f Cons[a Cons[b Nil]] r]

invoking a map object (with a *val*-labelled message containing a name f representing some function, a list Cons[a Cons[b Nil]], and a synchronization name r) translates into the process

new w Nil[w] | **new** v Cons[v b w] |
new u Cons[u a v] | map ! [f u r].

We can instantiate a Cell with a boolean value, result of checking whether a list l is empty,

Cell[c (**case** l **of** {Nil⇒True, Cons⇒(_ _) False})]

and translate it into the process

new r l![r] | r?{*Nil*= **new** v True[v] | Cell[c v],
Cons= (_ _) **new** v False[v] | Cell[c v]}.

Notice how expressions mix well with processes, even in the case where processes were not designed with expressions in mind.

The type system for expressions can be incorporated in the type system $\mathrm{TA}_\forall$ of core-TyCO through a simple adaptation of rules OBJ, MSG, and APP. The *type system $TA_{\forall exp}$ for processes with expressions* is obtained from the polymorphic type system $\mathrm{TA}_\forall$ by replacing axiom MSG, and rules OBJ and APP by the rules in figure 6, by replacing statements $B,\Gamma \vdash P$ by $D, B,\Gamma \vdash P$ (and similarly for abstractions and methods) in the remaining rules of $\mathrm{TA}_\forall$.

For example, using the MSGEXP-rule and the deduction for the expression Cons[a Cons[b Nil]] in section 5, we can show that the following statement is deducible in system $\mathrm{TA}_{\forall\mathrm{exp}}$.

$$B_0, \{\mathsf{map} : \{\mathit{val} : \alpha\, \mathsf{boolList}\, \beta\}, \mathsf{f} : \alpha, \mathsf{r} : \beta, \mathsf{a} : \mathsf{bool}, \mathsf{b} : \mathsf{bool}\} \vdash \mathsf{map}\,!\,[\mathsf{f}\ \mathsf{Cons}[\mathsf{a}\ \mathsf{Cons}[\mathsf{b}\ \mathsf{Nil}]]\ \mathsf{r}]$$

Similarly, using the APPEXP-rule and the deduction for the expression **case** l **of** {Nil⇒ True, Cons⇒ (_ _) False} in section 5, we can show that the following statement is deducible in system $\mathrm{TA}_{\forall\mathrm{exp}}$,

$$B_0, \{c : \{\mathit{read} : \mathsf{bool}, \mathit{write} : \{\mathit{val} : \mathsf{bool}\}\}, \mathsf{l} : \mathsf{boolList}\} \vdash \mathsf{Cell}[\mathsf{c}\ (\textbf{case}\ \mathsf{l}\ \textbf{of}\ \{\mathsf{Nil}\Rightarrow\mathsf{True}.\ \mathsf{Cons}\Rightarrow(\_\ \_)\ \mathsf{False}\})]$$

where we have added to B_0 the type assignment $\{\mathsf{Cell} : \forall t.(\{\mathit{read} : t, \mathit{write} : \{\mathit{val} : t\}\}t)\}$.

7. Expressions II - Abstraction and application

This section completes the presentation of expressions initiated in section 5, by introducing abstraction and application over expressions, as well as a form of recursion.

$$E \quad ::= \quad \cdots \quad | \quad \textbf{fn } \tilde{x} \Rightarrow l[\tilde{E}] \quad | \quad E\,!\,l[\tilde{E}] \quad | \quad \textbf{rec } x.E$$

Intuitively, an *abstraction expression* **fn** $\tilde{x} \Rightarrow l[\tilde{E}]$ is a (functional) object that, when invoked, returns an l-labelled message carrying the result of evaluating expressions $\tilde{E}$. The complimentary *application expression* $E\,!\,l[\tilde{E}]$ denotes the result of invoking an object located at (the result of evaluating) E via a l-labelled message with (the result of evaluating) expressions $\tilde{E}$. Expressions of the form **rec** $x.E$ (E must be an abstraction) define recursive functions when x occurs free in E.

$$\textsc{MsgExp}\quad \frac{B,\Gamma\vdash E:\alpha \qquad B,\Gamma\vdash \tilde{E}:\tilde{\alpha}_i}{D,B,\Gamma\vdash E\,!\,l_i[\tilde{E}]} \quad (\alpha\approx\{l_1:\tilde{\alpha}_1,\dots,l_n:\tilde{\alpha}_n\})$$

$$\textsc{ObjExp}\quad \frac{B,\Gamma\vdash E:\alpha \qquad D,B,\Gamma\vdash M:\alpha}{D,B,\Gamma\vdash E?M} \qquad\qquad \textsc{AppExp}\quad \frac{D,B,\Gamma\vdash U:\tilde{\alpha} \qquad B,\Gamma\vdash \tilde{E}:\tilde{\alpha}}{D,B,\Gamma\vdash U[\tilde{E}]}$$

FIG. 6. New rules for the type system of processes with expressions.

The mapping for the new expressions into the core calculus is defined as follows.

$$\begin{aligned}
[\![\textbf{fn}\ \tilde{x}\Rightarrow l[\tilde{E}]]\!]_v P &\stackrel{\text{def}}{=} \textbf{new}\ v\ \{\!|v:=\textbf{fn}\ \tilde{x}\Rightarrow l[\tilde{E}]|\!\}\mid P\\
[\![E\,!\,l[\tilde{E}]]\!]_v P &\stackrel{\text{def}}{=} \textbf{let}\ v=E\,!\,l[\tilde{E}]\ \textbf{in}\ P\\
[\![\textbf{rec}\ x.E]\!]_v P &\stackrel{\text{def}}{=} [\![E[v/x]]\!]_v P\\
\{\!|v:=\textbf{fn}\ \tilde{x}\Rightarrow l[\tilde{E}]|\!\} &\stackrel{\text{def}}{=} v?*\{(\tilde{x}r)[\![\tilde{E}]\!]_{\tilde{v}}(r\,!\,l[\tilde{v}])\}\\
\{\!|\textbf{let}\ x=E\,!\,l[\tilde{E}]\ \textbf{in}\ P|\!\} &\stackrel{\text{def}}{=} \textbf{new}\ r\ [\![E\tilde{E}]\!]_{v\tilde{v}}(v\,!\,l[\tilde{v}r])\mid\\
&\qquad r?\{(x)P\}
\end{aligned}$$

Application expressions use a new process constructor **let** $x = E\,!\,l[\tilde{E}]$ **in** P.[3] Such a process evaluates E into v, $\tilde{E}$ into $\tilde{v}$, emits a message $v\,!\,l[\tilde{v}r]$ with the synchronization name r, and then waits at r for a *val*-labelled reply.

Suppose cell is the location of a one-place buffer Cell holding a boolean value. We can negate its value by issuing a message not![cell!*read*] to be translated into the process

let v = cell!*read* **in** not![v]

or

new r cell!*read*[r] | r?{(v) not![v]}.

An abstraction **fn** $\tilde{x}\Rightarrow l[\tilde{E}]$ is encoded into a name v in a context P as the parallel composition of an environment entry $\{\!|v:=\textbf{fn}\ \tilde{x}\Rightarrow l[\tilde{E}]|\!\}$ and the process P itself, under a **new** v (cf. constructed values $C[\tilde{E}]$ in section 5). An environment entry of the above form is a replicated object with a single *val*-labelled method. To encode synchronous method invocation, the method receives not only the actuals for $\tilde{x}$ but also a reply-to name r. The body of the method evaluates the expressions $\tilde{E}$ into names $\tilde{v}$ and replies $r\,!\,l[\tilde{v}]$.

If name l represents a list of boolean values, we can negate every element in l by sending a *val*-labelled message to some object located at name map as in

map![(**fn** x ⇒ **if** x **then** False **else** True) l r].

Such a message is encoded into a v-located functional object and a message map![v l r] as follows.

new v v?*{(x r)= **new** r' x![r'] |
        r'?{*True*= **new** v False[v] | r![v],
            False= **new** v True[v] | r![v]}} |
    map![v l r])

Recursive expressions **rec** $x.E$ allow E to directly invoke the (replicated) object encoding the recursive expression. To apply the factorial function to each element in a list l, we may write the following piece of code. We leave the encoding for the reader.

apply![(**rec** f. **fn** x ⇒ **if** x=0 **then** 1 **else** x*f![n-1]) l]

Figure 5 shows the typing rules for the new expressions. To make clear the applicative behaviour of name-expressions, abbreviate type $\{\mathit{val}:\tilde{\alpha}\{\mathit{val}:\beta\}\}$ to $\tilde{\alpha}\to\beta$. Then the particular form of rules ExpAbs and ExpApp for expressions **fn** $\tilde{x}\Rightarrow[E]$ and $E\,!\,[\tilde{E}]$ become the following.

$$\frac{B,\Gamma\cdot\tilde{x}:\tilde{\alpha}\vdash E:\beta}{B,\Gamma\vdash(\textbf{fn}\ \tilde{x}\Rightarrow[E]):\tilde{\alpha}\to\beta}$$

$$\frac{B,\Gamma\vdash E:\tilde{\alpha}\to\beta \quad B,\Gamma\vdash\tilde{E}:\tilde{\alpha}}{B,\Gamma\vdash(E\,!\,[\tilde{E}]):\beta}$$

For an example with application, a simple deduction ending with rule MsgExp shows that

$\{\mathsf{cell}:\{\mathit{read}:\mathsf{bool},\mathit{write}:\{\mathit{val}:\mathsf{bool}\}\},$
$\quad\mathsf{not}:\mathsf{bool}\to\mathsf{bool}\}\vdash$ not![cell!*read*] : bool.

Also, using rules Name, Con, Cond, and ExpAbs, we can easily see that

$B_0,\emptyset\vdash$ (**fn** x ⇒ **if** x **then** False **else** True) : bool → bool.

Then, using the MsgExp-rule and denoting by Γ the typing {l : boolList, r : {*val* : boolList}, map : {(bool → bool) boolList → boolList}}, we have that

$B_0,\Gamma\vdash$ map![(**fn** x ⇒ **if** x **then** False **else** True) l r].

8. Functional objects and branch statements

This section brings into the process world useful constructs used to encode expressions in sections 5 and 7, namely the environment entries (for constructing data and functions), and the let/branch constructs.

8.1. Environment entries

Environment entries for data construction and for functions can be used directly for programming processes. Recall their syntax and encodings, where, in the second line, v may occur free in $\tilde{E}$.[4]

$$\{\!|v := C[\tilde{E}]|\!\} \stackrel{\text{def}}{=} [\![\tilde{E}]\!]_{\tilde{v}} C[v\tilde{v}]$$
$$\{\!|v := \textbf{fn}\ \tilde{x} \Rightarrow l[\tilde{E}]|\!\} \stackrel{\text{def}}{=} v?*\{(\tilde{x}r)[\![\tilde{E}]\!]_{\tilde{v}}(r\,!\,l[\tilde{v}])\}$$

Environment entries for datatype constructors are useful for sharing data among concurrent processes. For example, for sharing a list Cons[a Cons[b Nil]] between processes P and Q we may write

new l l:=Cons[a Cons[b Nil]] $\mid P \mid Q$.

On the other hand, functional objects constitute a convenient means to implement recursive functions. The list map function can be written as follows.

```
map := fn f l ⇒ case l of
        {Nil⇒ Nil,
         Cons⇒ (head tail) Cons[f![head] map![f tail]]}
```

Figure 7 presents the typing rules for the new processes. Using these rules, we can easily show that

$$B_0, \mathsf{map} : (\mathsf{bool} \to \mathsf{bool})\,\mathsf{boolList} \to \mathsf{boolList} \vdash \mathsf{map} := \textbf{fn}\ \mathsf{f}\ \mathsf{l} \Rightarrow \dots$$

8.2. Branch and let statements

Branch and let statements were introduced to encode case and application expressions, respectively in sections 5 and 7. The branch statement may now be generalised to arbitrary expressions, turning the let process a particular case of this. The let itself is generalised to bind sequences of names, rather than a single name. The general forms are useful to implement synchronous method invocation.

The new syntax and encoding is as follows.

$$\{\!|\textbf{branch}\ E!l[\tilde{E}]\ \textbf{into}\ M|\!\} \stackrel{\text{def}}{=} \textbf{new}\ r\ \{\!|E\tilde{E}|\!\}_{u\tilde{u}}(u\,!\,l[\tilde{u}r]) \mid r?M$$
$$[\![\textbf{let}\ \tilde{x} = E\,!\,l[\tilde{E}]\ \textbf{in}\ P]\!] \stackrel{\text{def}}{=} \textbf{branch}\ E!l[\tilde{E}]\ \textbf{into}\ \{(\tilde{x}) = P\}$$

For example, the process that branches into P or Q according to the result of a pop operation on a stack can we written as follows.

branch stack!*pop* **into** $\{$*val*$=$ (x)P, *empty*$=$ $Q\}$

The typing rule for branch is shown in figure 7; the rule for let is just a particular case; we leave it to the reader.

For example, assuming that Γ is a typing associating the type $\{\mathit{push} : \mathsf{nat}, \mathit{pop} : \{\mathit{val} : \mathsf{nat}, \mathit{empty}\}\}$ with name stack, and that $B, \Gamma \vdash P \mid Q$ for some basis B, by using rules Abs, Meth, and Branch we can easily see that

$$B, \Gamma \vdash \textbf{branch}\ \mathsf{stack}!\mathit{pop}\ \textbf{into}\ \{\mathit{val} = (\mathsf{x})P, \mathit{empty} : Q\}.$$

9. Main results

This section presents results that justify the type system for expressions and for processes extended with expressions.

The typing lemma for expressions is useful in establishing the main result.

Lemma 5. *If $B, \Gamma \vdash E : \alpha$, then $B, \Gamma \cdot x : \beta \vdash E : \alpha$.*

Proof A simple induction on the length of derivations. □

The type system for expressions is admissible

Lemma 6. *If $B, \Gamma \vdash E : \alpha$ and $B, \Gamma \cdot v : \alpha \vdash P$, then $B, \Gamma \vdash [\![E]\!]_v P$.*

Proof By induction on the length of the deduction of $B, \Gamma \vdash E : \alpha$.

When the deduction ends with axiom Name, the result follows by the substitution lemma 2.

When the deduction ends with rule Con, we know that $B, \Gamma \vdash C : \alpha\tilde{\alpha}$; hence that $B, \Gamma \cdot v\tilde{v} : \alpha\tilde{\alpha} \vdash C[v\tilde{v}]$. Using the induction hypothesis as many times as there are expressions in $\tilde{E}$, we get $B, \Gamma \cdot v; \alpha \vdash [\![\tilde{E}]\!]_{\tilde{v}}(C[v\tilde{v}])$. By hypothesis, $B, \Gamma \cdot v : \alpha \vdash P$. Use rules Par and Res.

When the deduction ends with the ExpAbs, let γ be the type $\{l : \tilde{\beta}, \dots\}$ and δ the type $\{\mathit{val} : \tilde{\alpha}\gamma\}$. Using the Msg-axiom we know that $B, \Gamma \cdot v\tilde{x}r\tilde{v} : \delta\tilde{\alpha}\gamma\tilde{\beta} \vdash r\,!\,l[\tilde{v}]$, and using the typing lemma for expressions 5 on the antecedent $B, \Gamma \cdot \tilde{x} : \tilde{\alpha} \vdash \tilde{E} : \tilde{\beta}$ of rule ExpAbs we get $B, \Gamma \cdot v\tilde{x}r : \delta\tilde{\alpha}\gamma \vdash \tilde{E} : \tilde{\beta}$. Then, by induction, followed by rules Abs, Meth, Obj, and Rep, we have $B, \Gamma \cdot v : \delta \vdash v?*\{(\tilde{x}r)[\![\tilde{E}]\!]_{\tilde{v}}(r\,!\,l[\tilde{v}])\}$. By hypothesis $B, \Gamma \cdot v : \delta \vdash P$, and the result follows from the application of rules Par and Res.

The three remaining cases are similar. □

The main result states that the type system for processes with expressions is admissible.

Theorem 7. *If $B, \Gamma \vdash_{\forall exp} P$, then $B, \Gamma \vdash \{\!|P|\!\}$.*

$$\text{ENV}\ \frac{D,B,\Gamma \vdash E : \alpha}{D,B,\Gamma \vdash a := E}\ (\Gamma(a) \approx \alpha) \qquad \text{BRANCH}\ \frac{B,\Gamma \vdash E : \{l : \tilde{\alpha}\beta, \dots\} \qquad B,\Gamma \vdash \tilde{E} : \tilde{\alpha} \qquad D,B,\Gamma \vdash M : \beta}{D,B,\Gamma \vdash \textbf{branch } E\,!\,l[\tilde{E}] \textbf{ into } M}$$

FIG. 7. Typing rules for the new processes.

PROOF By induction on the length of deductions. The interesting cases are when deductions end with the rules DTYPE, APPEXP, MSGEXP, and OBJEXP.

When the deduction ends with rule DTYPE, we have a process of the form

$$\textbf{datatype } T_1 = \{C_{11} : \tilde{T}_{11}, \dots, C_{1k_1} : \tilde{T}_{1k_1}\} \textbf{ and} \cdots \textbf{ and } \{C_{n1} : \tilde{T}_{n1}, \dots, C_{nk_n} : \tilde{T}_{nk_n}\} \textbf{ in } P.$$

For each $i = 1 \dots n$ and $j = 1 \dots k_i$, let γ_i be the type $\{C_{i1} : \tilde{\beta}_{i1}, \dots, C_{ik_i} : \tilde{\beta}_{ik_i}\}$, and let α_i be the type $\mu t.\{\mathsf{val} : \gamma_i\}$. Let B' be the basis $B \cdot C_{11} : (\alpha_1\tilde{\beta}_{11}) \cdots C_{nk_n} : (\alpha\beta_{nk_n})$. Using the MSG-axiom we know that $B', \Gamma \cdot c\tilde{x}_{ij}r : \alpha\tilde{\beta}_{ij}\gamma_i \vdash r\,!\,C_{ij}[\tilde{x}_{ij}]$. Then, using rules ABS, METH, OBJ, REP, and ABS, we get $B', \Gamma \vdash \{\!\!\{C_{ij} : \tilde{T}_{ij}\}\!\!\} : \alpha\tilde{\beta}_{ij}$. Using the induction hypothesis on the antecedent $B', \Gamma \vdash P$ of rule DTYPE, we have that $B', \Gamma \vdash \{\!\!\{P\}\!\!\}$. The result follows by the DEF-rule.

When the deduction ends with rule APPEXP, starting from the antecedent $B, \Gamma \vdash U : \tilde{\alpha}$ of the rule, and using rule APP and the typing lemma 1, we have $B, \Gamma \cdot \tilde{v} : \tilde{\alpha} \vdash U[\tilde{v}]$. The result follows from the antecedent $B, \Gamma \vdash \tilde{E} : \tilde{\alpha}$ of rule APP, and lemma 6.

When the deduction ends with rule OBJEXP, we first establish that

$$\text{if } B, \Gamma \vdash_{\forall \text{exp}} M, \text{ then } B, \Gamma \vdash \{\!\!\{M\}\!\!\}$$

using rules METH, ABS, and the induction hypothesis. Then, by rule OBJ and the typing lemma 1 we know that $B, \Gamma \cdot v : \alpha \vdash v?M$. The result follows from the antecedent $B, \Gamma \vdash E : \alpha$ of rule OBJ and lemma 6.

The APPEXP case is similar. □

10. Related work

The project closest to TyCO is Pict [19], a programming language based on the π-calculus [13]. Complex values in Pict include abstraction and application (cf. section 7), as well as a value constructor allowing to declare new names. No form of datatype declaration (cf. section 4), or value construction or de-construction (cf. section 5) is present in Pict.

Values in Pict are active: to get the value x of a complex value associated with a channel c, we just wait on c with an object $c?(x).P$. The strategy followed in this paper is slightly different. Expressions are encoded into *passive* data located at some name v. To get the value of the expression, we evaluate v (with a *val*-labelled message), and wait for the reply, as in the process **let** $x = v!$ **in** P. Clearly, when compared to TyCO, a straightforward implementation of complex values in Pict generates, for each name in the expression, one extra name (that may be later object of garbage collection) and one extra message.

Type abbreviations in Pict are parametric, allowing to declare, for example, a list of elements of an arbitrary type. For simplicity datatype declarations in this paper do not allow parametric types; their inclusion should not present major obstacles. On the other hand, Pict introduces no notion of datatype declarations. For example, the programmer is supposed to declare an abbreviation for the list type, as well as the constructors for Nil and the Cons (cf. section 4).

Pict's type system incorporates notions of subtyping and impredicative polymorphism [4, 15], thus being more powerful than the essentially Damas-Milner [6] type system of TyCO. As such, Pict demands explicit type annotations on bound variables for there is no algorithm which can infer all necessary type annotations. Pict's type system also distinguishes different usages (input, output, or both) of channels [18], thus opening the possibility of compiler optimizations.

While it is possible to encode TyCO into the π-calculus (and vice-versa) [28], labelled sums yield a much more elegant and efficient description of methods, with a direct counterpart in the type system. Pict implements each method in an object with a different server. A client of an object must then possess a different name for each different method. The execution of methods in mutual exclusion is achieved by a lock, implemented as a message containing the next state of the object. As such, for a method to execute, it must get not only an invocation, but also the lock (that is, the object's state). This mechanism is embellished with a deep layer of syntatic sugar, but the mutual exclusion via locks must be explicitly programmed.

Walker showed how to describe a semantics for concurrent object-oriented programming languages by a systematic translation of a (subset) of the POOL [3] programming language into the π-calculus [29]. Each method in an object is assigned a different name; all these names are multiplexed into a single name, making the invocation of a method a two-way protocol, against a one-way in TyCO.

TyCO has the Actor model of computation [2, 9] among its most influential sources. Back in 1977, Hewitt [8] already proposes the idea of building high level constructors on top a basic concurrent model.

There is one limitation to the predicative polymorphism induced by process abstraction [24]: although process abstractions may be polymorphic, names remain monomorphic. As such, for example, we cannot have a generic forwarder $a?*\{(x)b\,!\,[x]\}$ (of *val*-messages from a name a to a name b) polymorphic on the type of the names x forwarded (let alone polymorphic on the label of the forwarded messages.) The particular form of declarations of the Join calculus [7] allows for typing such a process, but on the other hand does not allow the generalization of a type variable if it appears free in more than one type, and, as such, is unable to type polymorphically the Cell example of section 2 (cf. section 3).

Oz [22] is a programming language that incorporates functional, object-oriented, and concurrent constraint programming by translation into a core calculus [21]. Although the calculus is based on concurrent constraints, Oz shares with Pict and TyCO the idea of building high-level constructors on top of a theoretical calculus.

Abadi and Cardelli proposed a basic calculus supporting builtin objects, method invocation, and method override [1]. Although concurrency is not present, the calculus shares with TyCO the notions of primitive objects composed of labelled methods, and that of labelled method invocation.

11. Further issues

Much remains to be done in the field of programming idioms for interaction. Among the most immediate developments to the material here proposed, are the following.

The incorporation of parameterized datatype declarations so that we can, for example, declare a list with elements of an arbitrary type as in

datatype list = (t) {Nil, Cons: t list[t]}.

The inclusion of patterns so that we can write

```
map?*{f Nil⇒ Nil,
      f Cons(head tail)⇒ Cons[f![head] map![f tail]]},
```

instead of the more verbose form map?*{**fn** f l⇒ **case** l **of** ...} of section 8.

Classes and subclasses are notions central in object-oriented programming. While TyCO provides for a simple notion of classes as templates from which objects may be instantiated, no notion of subclassing is present. The original version of this paper [28] contemplates a notion of behaviour subclassing that copies (at declaration or instantiation time) the code of the superclass into the subclass, thus avoiding dynamic lookup of methods and making the executable objects self contained. It does not however provide for separate compilation.

The interplay between processes and expressions is delicate. The present work achieves a good balance by allowing any non-binding occurrence of a name in the core-calculus to be a substituted by a general expression. It is our firm conviction that the encoding of expressions presented behave in a call by value [20] fashion; details remain to be checked.

Notes

1. In kinded type systems [17, 28] abstraction Cell is assigned a type $(\{\mathit{write} : \alpha, \mathit{read} : t'\}, t)$, and type variable t' is assigned a kind $\{\mathit{val} : \alpha\}$. Actual types for Cell are obtained by replacing t' by records containing at least the component $\mathit{val} : \alpha$ in the kind.
2. The usage of the wildcards in binding positions is conventional: $(\_)P$ denotes the abstraction $(x)P$ for some x not free in P
3. The let construct is inspired in the Pict programming language. Pict allows only names in the application expression, that is, it must be of the form $f\,!\,\mathit{val}[\tilde{a}]$ ($f\,!\,[\tilde{a}]$ for short), written in Pict as $f(\tilde{a})$.
4. Name v occurs free in E when this comes from a recursive definition (cf. section 7.)

References

[1] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data-Types*, volume 173 of *LNCS*. Springer-Verlag, 1984.

[5] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In *Logic and Computer Science*, pages 19–75. Academic Press limited, 1990.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[7] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[8] Carl Hewitt. Control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[9] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[10] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[11] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language. Version 1.2. *ACM Sigplan Notices*, 27(5), 1992.

[12] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised).* MIT Press, 1997.

[15] John C. Mitchell. *Theoretical aspects of object-oriented programming: types, semantics, and language design*, chapter Toward a typed foundation for method specialization and inheritance. MIT Press, 1994.

[16] A. Mycroft. Polymorphic type schemes for functional programming. In *9th ACM Symposium on Principles of Programming Languages*. ACM Press, 1984.

[17] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *19th ACM Symposium on Principles of Programming Languages*, pages 154–165. ACM Press, 1992.

[18] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *8th IEEE Symposium on Logic in Computer Science*, June 1993.

[19] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1997.

[20] G.D. Plotkin. Call-by-name and call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] Gert Smolka. A foundation for concurrent constraint programming. In *Constraints in Computational Logics*, volume 845 of *LNCS*. Springer-Verlag, March 1994.

[22] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.

[23] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP International Conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

[24] Vasco T. Vasconcelos. Predicative polymorphism in π-calculus. In *6th Parallel Architectures and Languages Europe*, volume 817 of *LNCS*, pages 425–437. Springer-Verlag, July 1994.

[25] Vasco T. Vasconcelos. *A process-calculus approach to typed concurrent objects.* PhD thesis, Keio University, 1994.

[26] Vasco T. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.

[27] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, August 1993.

[28] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st ISOTAS*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

[29] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.