TyCO Abstract Machine
– The Definition –

Luís Lopes Vasco T. Vasconcelos

Technical Report Series: DCC-97-1

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

TyCO Abstract Machine – The Definition *

Luís Lopes
Faculdade de Ciências, Universidade do Porto
e-mail: lblopes@ncc.up.pt

Vasco T. Vasconcelos
Faculdade de Ciências, Universidade de Lisboa
e-mail: vv@di.fc.ul.pt

April 1997

Abstract

We present an abstract machine specification for TyCO (Typed Concurrent Objects), a name-passing calculus that aims at capturing fundamental concepts present in Concurrent Object-Oriented Languages.

TyCO has built-in labeled messages and ephemeral objects that communicate asynchronously. Persistent objects are supported through instantiation of recursive classes. Concurrency is pervasive and synchronous communication can be implemented by passing continuations in messages. Strong, static typing is provided by a type inference algorithm that supports a form of predicative polymorphism.

This paper presents the formal framework of the TyCO abstract machine.

Keywords: Object-Oriented, Concurrency, Language, Implementation

1 Introduction

Name-passing process calculi attempt to formally describe the interaction between concurrent processes. This interaction involves not only point-to-point communication but also the notions of process mobility and communication topology. Milner, Parrow and Walker's π-calculus [6] and an asynchronous form due to Honda and Tokoro [4]. Have provided the basis for most current pure research on concurrency.

In recent years researchers have devoted a lot of effort in providing semantics for objects within the asynchronous π-calculus [4, 10, 7, 5], variations [8, 2] and extensions [3].

TyCO [8, 9] is just one of these efforts. It is a calculus that formally describes the concurrent interaction of ephemeral objects through asynchronous communication. The calculus grows from the polyadic π-calculus and its monomorphic type system, by introducing recursive types, a polymorphic type system [9], templates and variables over templates. Synchronous communication can be achieved as usual by incorporating in messages an extra name intended to convey the result of the communication. The fundamental entities of the calculus are messages and ephemeral objects. Templates are specifications of processes abstracted on a sequence of variables. They allow classes to be modeled from which run-time objects may be instantiated.

*This work was partially supported by the Dolphin project (contract PRAXIS/2/2.1/TIT/1577/95).

The abstract machine evolves from the object calculus by defining a static semantics (a type assignment system) and a dynamic semantics in the form of a set of reduction rules. The abstract syntax of the calculus also evolves into a more compact object *assembly language*.

The remainder of the paper is organized as follows: section 2 describes the abstract syntax of the machine *assembler*; section 3 describes the machine components and its reduction rules and finally, section 4 introduces the type assignment system we developed for machine programs.

2 Abstract Syntax

2.1 Sequences and finite maps

1. $\vec{e}$ denotes the sequence $e_1 \ldots e_k$ $(k \geq 0)$ of elements of some syntactic category, and $\{\vec{e}\}$ denotes the set $\{e_1, \ldots, e_n\}$. The empty sequence is denoted by ϵ.

2. When A and B are sets, $A \mapsto B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B.

3. A finite map is often written explicitly in the form $\{a_1 : b_1, \ldots, a_k : b_k\}$ for $k \geq 0$. In particular, the empty map is $\{\}$. When $\vec{a} = a_1 \ldots a_k$ and $\vec{b} = b_1 \ldots b_n$, we abbreviate the above finite map to $\vec{a} : \vec{b}$.

4. When f and g are finite maps, the map $f + g$, called f *modified* by g, is the finite map with domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and values

$$(f + g)(a) = \text{ if } a \in \mathrm{dom}(g) \text{ then } g(a) \text{ else } f(a).$$

5. When f is a finite map and V a set, the map $f \setminus V$, is the finite map with domain $\mathrm{dom}(f) \setminus V$ and values

$$(f \setminus V)(a) = f(a).$$

6. When f is a finite map and V a set, the map $f \restriction V$, pronounced f *restricted to* V, is the finite map with domain $\mathrm{dom}(f) \cap V$ and values

$$(f \restriction V)(a) = f(a).$$

2.2 Identifiers and Syntactic Categories

$c \in \mathrm{Const}$	Constants (Integer and Boolean)
$x, y, z \in \mathrm{Var}$	Variables
$u, v \in \mathrm{Var} \cup \mathrm{Const}$	Values
$l \in \mathrm{Label}$	Labels
$X \in \mathrm{CVar}$	Class variables
$P \in \mathrm{Proc}$	Processes
$\vec{P} \in \mathrm{Thread} = \mathrm{Proc}^*$	Threads
$M \in \mathrm{Meth} = \mathrm{Label} \mapsto \mathrm{Templ}$	Methods
$A \in \mathrm{Templ} = \mathrm{Var}^* \times \mathrm{Thread}$	Templates
$C \in \mathrm{Class} = \mathrm{Var}^* \times \mathrm{Var} \times \mathrm{Meth}$	Classes
$D \in \mathrm{DBind} = \mathrm{CVar} \mapsto \mathrm{Class}$	Class Bindings

2.3 Syntax

$$
\begin{array}{rcl}
P & ::= & u \lhd l : [\vec{u}] \mid y \rhd M \mid \ \mid x \mid \vec{P} \mid X[\vec{u}] \mid \mathsf{def}\ D\ \mathsf{in}\ \vec{P} \\
\vec{P} & ::= & P_1 \mid \ldots \mid P_k \\
M & ::= & \{l_1 : A_1\ , \ldots,\ l_k : A_k\} \\
A & ::= & (\vec{x})\vec{P} \\
D & ::= & \{X_1 : C_1\ , \ldots,\ X_k : C_k\} \\
C & ::= & (\vec{x})y \rhd M
\end{array}
$$

The sequence $\vec{P} = P_1 \mid \ldots \mid P_k$ for $k = 0$ is designated by the keyword nil.

2.4 Syntactic conventions

- The names in the sequence $\vec{x}$ in a template A or a class C are pairwise distinct.
- The set of methods M for an object is written as $\{l_1 : A_1\ ; \ldots ;\ l_k : A_k\}$ in abstract machine programs.
- The set of class bindings $\{X_1 : C_1\ , \ldots,\ X_k : C_k\}$ in a def statement is written as $X_1 = C_1\ \mathsf{and} \ldots \mathsf{and}\ X_k = C_k$ in abstract machine programs.

3 The Abstract Machine

3.1 Queues

Queues are used to implement the run-queue as well as the messages/object queues; hence the parametric definition.

$$
\mathrm{Queue}(A) \quad ::= \quad A :: \ldots :: A \quad \mid \quad \bullet
$$

3.2 Free identifiers

Identifiers take the form of value- and class-variables. These may occur free or bound in processes.

The set of *free value-variables* in a process is inductively defined by:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{fv}(c) & = & \emptyset & \mathrm{fv}(u \lhd l : [\vec{u}]) & = & \mathrm{fv}(u) \cup \mathrm{fv}(\vec{u}) \\
\mathrm{fv}(x) & = & \{x\} & \mathrm{fv}(y \rhd M) & = & \mathrm{fv}(y) \cup \mathrm{fv}(M) \\
\mathrm{fv}(\mid x \mid \vec{P}) & = & \mathrm{fv}(\vec{P}) \setminus \{x\} & \mathrm{fv}(M) & = & \cup_{l \in \mathrm{dom}(M)}\, \mathrm{fv}(M(l)) \\
\mathrm{fv}(P_1 \mid \ldots \mid P_k) & = & \mathrm{fv}(P_1) \cup \ldots \cup \mathrm{fv}(P_k) & \mathrm{fv}(D) & = & \cup_{X \in \mathrm{dom}(D)}\, \mathrm{fv}(D(X)) \\
\mathrm{fv}(X[\vec{u}]) & = & \mathrm{fv}(\vec{u}) & \mathrm{fv}((\vec{x})\vec{P}) & = & \mathrm{fv}(\vec{P}) \setminus \{\vec{x}\} \\
\mathrm{fv}(\mathsf{def}\ D\ \mathsf{in}\ \vec{P}) & = & \mathrm{fv}(D) \cup \mathrm{fv}(\vec{P}) & & &
\end{array}
$$

The set of *bound value-variables* is defined accordingly. A process P is said to be *closed for value-variables* if $\mathrm{fv}(P) = \emptyset$.

The set of *free class-variables* in a process is inductively defined by:

$$\begin{aligned}
\mathrm{fc}(c) &= \emptyset & \mathrm{fc}(u \lhd l : [\vec{u}]) &= \emptyset \\
\mathrm{fc}(x) &= \emptyset & \mathrm{fc}(y \rhd M) &= \mathrm{fc}(M) \\
\mathrm{fc}(|\, x \,|\, \vec{P}) &= \mathrm{fc}(\vec{P}) & \mathrm{fc}(M) &= \cup_{l \in \mathrm{dom}(M)} \mathrm{fc}(M\,(l)) \\
\mathrm{fc}(P_1 | \ldots | P_k) &= \mathrm{fc}(P_1) \cup \ldots \cup \mathrm{fc}(P_k) & \mathrm{fc}(D) &= (\cup_{X \in \mathrm{dom}(D)} \mathrm{fc}(D(X))) \setminus \mathrm{dom}(D) \\
\mathrm{fc}(X[\vec{u}]) &= \{X\} & \mathrm{fc}((\vec{x})\vec{P}) &= \mathrm{fc}(\vec{P}) \\
\mathrm{fc}(\mathsf{def}\ D\ \mathsf{in}\ \vec{P}) &= \mathrm{fc}(D) \cup (\mathrm{fc}(\vec{P}) \setminus \mathrm{dom}(D))
\end{aligned}$$

The set of *bound class-variables* is defined accordingly. A process P is said to be *closed for class-variables* if $\mathrm{fc}(P) = \emptyset$.

3.3 Identifiers and Syntactic Categories

$a \in \mathrm{Name}$	Names
$v \in \mathrm{Val} = \mathrm{Name} \cup \mathrm{Const}$	Values
$B \in \mathrm{VBind} = \mathrm{Var} \mapsto \mathrm{Val}$	Variable bindings
$T \in \mathrm{TBind} = \mathrm{CVar} \mapsto (\mathrm{VBind} \times \mathrm{Class})$	Class-variable bindings
$ms \in \mathrm{QComm} = \mathrm{Queue}(\mathrm{Label} \times \mathrm{Val}^*)$	Communication queue
$Ms \in \mathrm{QMeth} = \mathrm{Queue}(\mathrm{VBind} \times \mathrm{Meth})$	Method-closure queue
$Q \in \mathrm{QBind} = \mathrm{Name} \mapsto (\mathrm{QComm} \cup \mathrm{QMeth})$	Communication/Method-closure queues
$R \in \mathrm{RunQueue} = \mathrm{Queue}(\mathrm{VBind} \times \mathrm{Thread})$	RunQueue
$S \in \mathrm{State} = \mathrm{TBind} \times \mathrm{VBind} \times \mathrm{QBind} \times \mathrm{Thread} \times \mathrm{RunQueue}$	Machine state

Variable bindings B are used with domain $\mathrm{Var} \cup \mathrm{Const}$. It follows that:

$$B(u) = \begin{cases} u & \text{if } u \in \mathrm{Const} \\ v & \text{if } u : v \in B \end{cases}$$

3.4 Restrictions on the Initial Process

1. The initial process is closed for class-variables and value-variables.
2. All bound variables in the initial process are distinct.
3. The initial process is well typed (see section 4).

3.5 The initial and the final state

The abstract machine starts computing with an empty run-queue, no bindings in T and B and no queues in Q. Thus the initial state is:

$$\emptyset, \emptyset, \emptyset, \vec{P}, \bullet$$

where $\vec{P}$ is the initial thread. The machine halts when no rule can be applied, that is when a state of the form below is reached.

$$_, _, _, \mathsf{nil}, \bullet$$

3.6 Reduction

Rules transform states into states. In the sequel, comments accompanying rules indicate conditions that must be met for the reductions to be successful. Failure to comply with any one of these conditions will deadlock the machine.

The SCHED rule detects the end of the current thread and starts executing the next one in the run-queue.

$$T, _, Q, \mathsf{nil}, (B, \vec{P}) :: R \longrightarrow T, B, Q, \vec{P}, R \qquad \text{(SCHED)}$$

The SCOP rule introduces a new local variable within a thread. For a process $|\, x \,|\, \vec{P}$, the rule adds a binding from variable x to a *fresh name* a, and creates a new empty queue for a in Q. Execution continues with thread $\vec{P}$.

$$\frac{a \notin \mathrm{dom}(Q)}{T, B, Q, |\, x \,|\, \vec{P} \mid \vec{P}', R \longrightarrow T, B + \{x : a\}, Q + \{a : \bullet\}, \vec{P} \mid \vec{P}', R} \qquad \text{(SCOP)}$$

Comment. $x \notin \mathrm{dom}(B)$ by the variable renaming assumption.
The DEF rule introduces new classes. Given a process $\mathsf{def}\ D\ \mathsf{in}\ \vec{P}$, for each definition $X = C$ in D, the rule adds a bind $X : (B, C)$. Execution continues with thread $\vec{P}$.

$$T, B, Q, \mathsf{def}\ D\ \mathsf{in}\ \vec{P} \mid \vec{P}', R \longrightarrow T + \{X : (B, C) \mid X = C \in D\}, B, Q, \vec{P} \mid \vec{P}', R \qquad \text{(DEF)}$$

Comment. $\mathrm{dom}(T) \cap \mathrm{dom}(D) = \emptyset$ by the variable renaming assumption. The RED rules are responsible for contraction of redexes. The REDMSG (REDOBJ) rule takes a message (an object) from the current thread, a matching object (message) from the appropriate object (message) queue, and place the body of the invoked method plus the variable bindings in the run-queue for later execution. The REDINST rule creates an object, instance of some class, and reduces it immediately with a compatible message. In all cases execution continues with the current thread.

$$\frac{B(y) = a \qquad Q(a) = (B', M) :: Ms \qquad M(l) = (\vec{x})\vec{P}'}{T, B, Q, y \lhd l : [\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : Ms\}, \vec{P}, R :: (B' + \{\vec{x} : B(\vec{u})\}, \vec{P}')} \qquad \text{(REDMSG)}$$

$$\frac{B(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{x})\vec{P}'}{T, B, Q, y \rhd M|\vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B + \{\vec{x} : \vec{v}\}, \vec{P}')} \qquad \text{(REDOBJ)}$$

$$\frac{T(X) = (B', (\vec{x})y \rhd M) \qquad B''(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{z})\vec{P}'}{T, B, Q, X[\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B'' + \{\vec{z} : \vec{v}\}, \vec{P}')} \qquad \text{(REDINST)}$$

where $B'' = B' + \{\vec{x} : B(\vec{u})\}$ in the REDINST rule.

Comment. $y \in \mathrm{dom}(B)$ since processes are closed.

Comment. $l \in \mathrm{dom}(M)$ since programs are well typed.

Comment. length($\vec{x}$)=length($\vec{v}$) since programs are well typed.

Comment. $a \in \mathrm{dom}(Q)$. Since processes are closed and well-typed, the binding must have been placed by the SCOP-rule.

Comment. $X \in \mathrm{dom}(T)$ since programs are closed for class-variables. Similarly, $\{\vec{u}\} \subseteq \mathrm{dom}(B)$.

Comment. $\{\vec{x}\} \cap \mathrm{dom}(B') = \emptyset$ since $\mathrm{dom}(B') = \mathrm{fv}((\vec{x})y \rhd M)$ and $\{\vec{x}\} \cap \mathrm{fv}(P') = \emptyset$.

The QUEUE rules are responsible for the queuing of messages and objects. They are used whenever immediate reduction of messages, objects or class instances is not possible. The QUEUEMSG (QUEUEOBJ) rule takes a message (an object) and place it in the appropriate message (object) queue. Reduction will occur when a matching object (message) is later scheduled for execution. Execution of the current thread continues. The QUEUEINST rule creates an object, instance of some class, and places it in the appropriate object queue.

$$\frac{B(y) = a \qquad Q(a) = ms}{T, B, Q, y \lhd l : [\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : ms :: l : [B(\vec{u})]\}, \vec{P}, R} \quad \text{(QUEUEMSG)}$$

$$\frac{B(y) = a \qquad Q(a) = Ms}{T, B, Q, y \rhd M \mid \vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B, M)\}, \vec{P}, R} \quad \text{(QUEUEOBJ)}$$

$$\frac{T(X) = (B', (\vec{x})y \rhd M) \qquad B'' = B' + \{\vec{x} : B(\vec{u})\} \qquad B''(y) = a \qquad Q(a) = Ms}{T, B, Q, X[\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B'', M)\}, \vec{P}, R} \quad \text{(QUEUEINST)}$$

Comment. $y \in \mathrm{dom}(B)$ since processes are closed.

Comment. $a \in \mathrm{dom}(Q)$. Since processes are closed and well-typed, the binding must have been placed by the SCOP-rule.

Comment. $X \in \mathrm{dom}(T)$ since programs are closed for class-variables. Similarly, $\{\vec{u}\} \subseteq \mathrm{dom}(B)$.

Comment. Packing variable bindings when making a closure for a thread or a set of methods, or de-referencing values when a thread is assigned for execution, is one major task of the abstract machine. These operations can be optimized both for space and speed.

First, when packing variable bindings into a method closure we do it per label, that is, instead of a single pair in VBind $\times$ Meth we use a collection of pairs in VBind $\times$ Templ:

$$\{l : (B \restriction \mathrm{fv}(A), A) \mid l : A \in M\}$$

at the cost of some duplication of bindings, this significantly decreases the average cardinality of the B component during computations. Bindings for the bound variables (parameters) are supplied when the thread is activated. For de-referencing values we use the following equality to avoid redundant operations:

$$B(u) = \begin{cases} u & \text{if } u \in \mathrm{Const} \\ v & \text{if } u : v \in B \end{cases}$$

To end this section we present a full listing of the rules for the abstract machine:

SCHED $$T, _, Q, \mathsf{nil}, (B, \vec{P}) :: R \longrightarrow T, B, Q, \vec{P}, R$$

SCOP $$\frac{a \notin \mathrm{dom}(Q)}{T, B, Q, |\, x \,|\, \vec{P} \mid \vec{P}', R \longrightarrow T, B + \{x : a\}, Q + \{a : \bullet\}, \vec{P} \mid \vec{P}', R}$$

DEF $$T, B, Q, \mathsf{def}\ D\ \mathsf{in}\ \vec{P} \mid \vec{P}', R \longrightarrow T + \{X : (B, C) \mid X = C \in D\}, B, Q, \vec{P} \mid \vec{P}', R$$

REDMSG $$\frac{B(y) = a \qquad Q(a) = (B', M) :: Ms \qquad M(l) = (\vec{x})\vec{P}'}{T, B, Q, y \lhd l : [\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : Ms\}, \vec{P}, R :: (B' + \{\vec{x} : B(\vec{u})\}, \vec{P}')}$$

REDOBJ $$\frac{B(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{x})\vec{P}'}{T, B, Q, y \rhd M \mid \vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B + \{\vec{x} : \vec{v}\}, \vec{P}')}$$

REDINST $$\frac{T(X) = (B', (\vec{x})y \rhd M) \qquad B''(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{z})\vec{P}'}{T, B, Q, X[\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B'' + \{\vec{z} : \vec{v}\}, \vec{P}')}$$

QUEUEMSG $$\frac{B(y) = a \qquad Q(a) = ms}{T, B, Q, y \lhd l : [\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : ms :: l : [B(\vec{u})]\}, \vec{P}, R}$$

QUEUEOBJ $$\frac{B(y) = a \qquad Q(a) = Ms}{T, B, Q, y \rhd M \mid \vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B, M)\}, \vec{P}, R}$$

QUEUEINST $$\frac{T(X) = (B', (\vec{x})y \rhd M) \qquad B'' = B' + \{\vec{x} : B(\vec{u})\} \qquad B''(y) = a \qquad Q(a) = Ms}{T, B, Q, X[\vec{u}] \mid \vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B'', M)\}, \vec{P}, R}$$

3.7 Basic Arithmetic and Logic Operations

The abstract machine supports two primitive data types: booleans and integers. Actually any of these data types could be encoded in TyCO [8] and their operations manipulated by the abstract machine just as conventional processes. However, performance constraints urged the definition of built-in operations (reduction rules) for these common data types.

The basic logic operations are provided by the following rules:

$$\begin{aligned}
T, B, Q, (c \lhd \mathsf{not} : [r]) \mid \vec{P}, R &\longrightarrow T, B, Q, (B(r) \lhd \mathsf{val} : [\neg B(c)]) \mid \vec{P}, R \\
T, B, Q, (c \lhd \mathsf{and} : [c'\ r]) \mid \vec{P}, R &\longrightarrow T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) \wedge B(c')]) \mid \vec{P}, R \\
T, B, Q, (c \lhd \mathsf{or} : [c'\ r]) \mid \vec{P}, R &\longrightarrow T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) \vee B(c')]) \mid \vec{P}, R \\
T, B, Q, (c \lhd \mathsf{xor} : [c'\ r]) \mid \vec{P}, R &\longrightarrow T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) \dot{\vee} B(c')]) \mid \vec{P}, R \\
T, B, Q, (c \lhd \mathsf{eq} : [c'\ r]) \mid \vec{P}, R &\longrightarrow T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) = B(c')]) \mid \vec{P}, R
\end{aligned}$$

Booleans can intuitively be seen as objects of the form below, one for each boolean c, permanently in the object-queue.

$$c \rhd \{\mathsf{not} : (r)\ r \lhd \mathsf{val} : [\neg c], \ldots\}$$

We also introduce the conditional construct if ... as a built-in both for performance and for keeping the design coherent with mappings of higher order constructs such as pattern matching [8]. The reduction rules for if c then $\vec{P}$ else $\vec{Q}$ (where c is a boolean constant) are the following:

$$\begin{aligned}
T, B, Q, \mathsf{if\ true\ then}\ \vec{P}'\ \mathsf{else}\ \vec{P}'' \mid \vec{P}, R &\longrightarrow T, B, Q, \vec{P}' \mid \vec{P}, R \\
T, B, Q, \mathsf{if\ false\ then}\ \vec{P}'\ \mathsf{else}\ \vec{P}'' \mid \vec{P}, R &\longrightarrow T, B, Q, \vec{P}'' \mid \vec{P}, R
\end{aligned}$$

Finally, the primitive integer operations are implemented by the following rules:

$$
\begin{array}{rcl}
T, B, Q, (c \lhd \mathsf{add} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) + B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{sub} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) - B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{mul} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) \cdot B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{div} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c)\ \mathrm{div}\ B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{mod} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c)\ \mathrm{mod}\ B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{eq} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) = B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{geq} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) >= B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{gt} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) > B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{leq} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) <= B(c')])\ |\vec{P}, R \\
T, B, Q, (c \lhd \mathsf{lt} : [c'\ r])\ |\vec{P}, R & \longrightarrow & T, B, Q, (B(r) \lhd \mathsf{val} : [B(c) < B(c')])\ |\vec{P}, R
\end{array}
$$

Integers can intuitively be seen as objects of the form below, one for each integer c, permanently in the object-queue.

$$c \rhd \{\mathsf{add} : (c'\ r)\ r \lhd \mathsf{val} : [c + c'], \ldots\}$$

4 Type Inference

This section presents a type system assigning types to the free names in a process.

Typable processes are exempt from run-time errors. We say a process P contains a possible run-time error if the initial state $\emptyset, \emptyset, \emptyset, \vec{P}, \bullet$ reduces to a state $T, B, Q, y \lhd l : \vec{u} \mid \vec{P}, R$ with $Q(a) = (B', M) :: Ms$ but the RedMsg rule cannot be applied because:

1. label l is not in M - the so called *message not understood* error. (similarly for the RedObj and RedInst rules);
2. although l is in M, and $M(l) = (\vec{x})\vec{P}$, the number of actuals in $\vec{x}$ does not match the number of forwards in $\vec{x}$ (similarly for the RedObj ,RedInst and QueueInst).

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 record and recursive type constructors. We use $t, t', \cdots$ to range over type variables, $\alpha, \beta, \cdots$ to range over types, and $\vec{\alpha}, \vec{\beta}, \cdots$ to represent (finite, possibly empty) sequences of types. The *set of monomorphic types* is given by the grammar:

$$\alpha \quad ::= \quad t \qquad | \qquad \{l_1 : \vec{\alpha_1}, \cdots, l_n : \vec{\alpha_n}\} \qquad | \qquad \mu t.\alpha$$

where $l_1, \cdots, l_n$ are pairwise distinct labels.

A simple extension of the monomorphic type system allows to abstract a type for an abstraction on a particular type variable, and to instantiate an abstracted type into different monomorphic types. Types for abstractions fall into two classes: *monomorphic types* as sequences of types for names, and *polymorphic types* constructed using $\forall$ [1]. The *set of polymorphic types* is given by the following grammar:

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

A type of the form $\{l_1 : \vec{\alpha_1}, \cdots, l_n : \vec{\alpha_n}\}$ is intended to describe some collection of names identifying objects containing n methods labeled with $l_1, \cdots, l_n$, and whose arguments of method l_i belong to the types in $\vec{\alpha_i}$. Types are interpreted as regular infinite trees. A type of the form $\mu t.\alpha$ (with $\alpha \neq t$ denotes the regular infinite tree solution of the equation $t = \alpha$. If α is a type, denote α^* its

associated regular infinite tree. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on recursive types, by putting $\alpha \approx \beta$ if $\alpha^* \approx \beta^*$.

Type assignments to names are formulae $x : \alpha$, for x a name and α a type. While we assign types to names, to processes we assign typings. *Typings*, denoted $\Gamma, \Delta, \cdots$, are sets of type assignments where no name occurs twice. The typing $\Gamma - \{\vec{x}\}$ is obtained from Γ by removing the type assignment associated with $\vec{x}$. The expression $\Gamma(x)$ denotes the type associated with x in Γ.

Universal quantifiers can only occur at the top level of types. This is a direct consequence of the definition of the set of polymorphic types. A polymorphic type of the form $\forall t_1 . \cdots \forall t_n . \vec{\alpha}$ is abbreviated to $\forall t_1 \cdots t_n . \vec{\alpha}$, where $t_1, \cdots, t_n$ are the bound variables of the type. (There is another binding for type variables, namely $\mu t . \alpha$ binds t in α.) 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 $\sigma[\alpha/t]$. A type variable occurs free in a typing Γ if it occurs free in some type in Γ.

Type assignments to abstraction variables are formulae $X : \sigma$, for X an abstraction variable and σ a polymorphic type. *Bases*, denoted $B, B', \cdots$, are sets of type assignments to variables where no variable occurs twice. Similarly to typings, we write $B(X)$ for the type associated with X in B. A type variable occurs free in a basis B if it occurs free in some type B.

When B is a base containing only sequences of types, the closure of B with respect to Γ, $\mathrm{clos}_\Gamma B$, is the basis $\{X : \forall \vec{t} . \vec{\alpha} \mid B(X) = \vec{\alpha}$ and $\vec{t}$ are the variables occurring free in $\vec{\alpha}$ but not in B or $\Gamma\}$.

Typing assignments to processes are statements $B, \Gamma \vdash P$, whereas *type assignment to abstractions* are statements $B, \Gamma \vdash A : \sigma$.

The *type system* is composed of the following axioms and rules:

Values are assigned types. We assume a predicate $typeof(c)$ for each constant c. Sequences of values are assigned sequences of types.

$$\Gamma \vdash c \Rightarrow \mathrm{typeof}(c) \qquad \frac{}{\Gamma, x : \alpha \vdash x \Rightarrow \alpha} \qquad \frac{\Gamma \vdash v \Rightarrow \alpha \qquad \Gamma \vdash \vec{v} \Rightarrow \vec{\alpha}}{\Gamma \vdash v\vec{v} \vdash \alpha\vec{\alpha}}$$

A *Method* $l : A$ is assigned a record type $\{l : \vec{\alpha}\}$ for $\vec{\alpha}$ the type of the formal parameters of the method.

$$\frac{B, \Gamma \vdash A \Rightarrow \vec{\alpha}}{B, \Gamma \vdash l : A \Rightarrow \{l : \vec{\alpha}\}} \qquad \frac{B, \Gamma \vdash M \Rightarrow \alpha \qquad B, \Gamma \vdash M' \Rightarrow \alpha'}{B, \Gamma \vdash M + M' \Rightarrow \alpha + \alpha'}$$

A *Communication* $l : \vec{v}$ is assigned a record type containing a component $l : \vec{\alpha}$, for $\vec{\alpha}$ the type of $\vec{v}$.

$$\frac{\Gamma \vdash \vec{v} \Rightarrow \vec{\alpha} \qquad \alpha'(l) = \vec{\alpha}}{B, \Gamma \vdash l : \vec{v} \Rightarrow \alpha'}$$

Abstractions $(\vec{x})P$ (that is, the body of a method or a class) is assigned the type of its formal arguments.

$$\frac{B, \Gamma \vdash P \Rightarrow \Gamma \qquad \Gamma \vdash \vec{x} \Rightarrow \vec{\alpha}}{B, \Gamma - \{\vec{x}\} \vdash (\vec{x})P \Rightarrow \vec{\alpha}}$$

A *Class Binding*, $X = C$ is assigned a base where the type $\vec{\alpha}$ of its formal parameters is assigned to X.

$$\frac{B, \Gamma \vdash C \Rightarrow \vec{\alpha}}{B, \Gamma \vdash X = C \Rightarrow \{X : \vec{\alpha}\}} \qquad \frac{B, \Gamma \vdash D \Rightarrow B \qquad B, \Gamma \vdash D' \Rightarrow B'}{B, \Gamma \vdash D + D' \Rightarrow B + B'}$$

As for *Processes*, a *message* $v \lhd l : \vec{v}$ is *ok* if the type of v contains a component $l : \vec{\alpha}$, for $\vec{\alpha}$ the type of $\vec{v}$. The type of an *object* $x \rhd M$ is the type of the methods M.

$$\frac{B, \Gamma \vdash \vec{v} \Rightarrow \vec{\alpha} \qquad \Gamma(v) = \{l : \vec{\alpha}, \cdots\}}{B, \Gamma \vdash v \lhd l : \vec{v} \Rightarrow ok} \qquad \frac{B, \Gamma \vdash M \Rightarrow \alpha \qquad \Gamma(x) = \alpha}{B, \Gamma \vdash x \rhd M \Rightarrow ok}$$

A *thread* is *ok* if every constituent process is *ok*. A *scope restriction* $|\, x \,|\, \vec{P}$ is *ok* if $\vec{P}$ is *ok*; since x is bound in $\vec{P}$ we remove its binding from the typing.

$$\frac{B, \Gamma \vdash P_1 \Rightarrow ok \;\; \cdots \;\; B, \Gamma \vdash P_k \Rightarrow ok}{B, \Gamma \vdash P_1 | \cdots | P_k, Q \vdash ok} \qquad \frac{B, \Gamma \vdash P \Rightarrow ok}{B, \Gamma - \{x\} \vdash |\, x \,|\, \vec{P} \Rightarrow ok}$$

A *class instantiation* is *ok* if the type for X is compatible with the type $\vec{\alpha}$ of the arguments $\vec{v}$. A def process is *ok* if $\vec{P}$ is *ok* with the types for the new class definitions added to base B.

$$\frac{B(X) \succ \vec{\alpha} \qquad \Gamma \vdash \vec{v} \Rightarrow \vec{\alpha}}{B, \Gamma \vdash X[\vec{v}] \Rightarrow ok} \qquad \frac{B, \Gamma \vdash D \Rightarrow B' \qquad B + \mathrm{clos}_\Gamma(B'), \Gamma \vdash P \Rightarrow ok}{B, \Gamma \vdash \mathsf{def}\ D\ \mathsf{in}\ \vec{P} \Rightarrow ok}$$

We conclude this section with a brief overview of some important properties of the type system. The free names in a process constitute the interaction points of the process, and the types of the names describe, in some sense, the interface to the process. Typings for typable processes assign types to all free names in the processes.

Subject reduction ensures that a typing for a process does not change as the process is reduced. As a corollary, typable processes do not encounter errors at runtime.

Decidability of typing assignment (given B, P, and Γ, is it the case that $B, \Gamma \vdash P$?) and *computability of typing existence* (given P, is there a B, Γ such that $B, \Gamma \vdash P$?) are two properties verified by the system despite the fact that the system possess no simple notion of subsumption (and hence of principal typings).

References

[1] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[2] G. Boudol. The π-calculus in Direct Style. In *Proceedings of POPL'97*. Springer-Verlag, January 1997.

[3] G. Smolka. A foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994. Invited Talk.

[4] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS, Springer-Verlag*, pages 141–162, 1991.

[5] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In M. Tokoro and O. Niestrasz and P. Wegner, editor, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, number 612 in Lecture Notes in Computer Science, pages 53–79. Springer-Verlag, 1992.

[6] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, (100):1–77, 1992.

[7] Oscar Niestrasz. Towards an Object Calculus. In M. Tokoro and O. Niestrasz and P. Wegner, editor, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, number 612 in Lecture Notes in Computer Science, pages 1–20. Springer-Verlag, 1992.

[8] V. Vasconcelos. Typed Concurrent Objects. In *8th European Conference on Object-Oriented Programming, LNCS*, volume 821, pages 100–117. Springer-Verlag, July 1994.

[9] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *1st International Symposium on Object Technologies for Advanced Software, LNCS*, volume 742, pages 460–474. Springer-Verlag, November 1993.

[10] David Walker. π-Calculus Semantics of Object-Oriented Programming Languages. In *Theoretical Aspects of Computer Software*, volume 526, pages 532–547, 1992.