An Abstract Machine for an Object Calculus

Luís Lopes Vasco T. Vasconcelos

Technical Report Series: DCC-97-5

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

An Abstract Machine for an Object Calculus *

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

July 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 object s 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 describes the abstract machine framework in detail, including how it evolves from TyCO and some important properties it can be shown to possess.

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. These calculi originate in Milner's CCS [13] (Calculus of Communicating Processes) in which processes interacted through synchronization (the simplest form of communication). π-calculus [15] was the first calculus in which processes could send one *upright* at a time through a named channel. This is called the monadic π-calculus. Later on this calculus was extended to allow for tuples to be exchanged in a single message, giving rise to the polyadic π-calculus [14]. This extension eased the programming task but did not increase the power of the calculus itself. Instead it introduced the possibility of protocol errors. These occur when two processes communicate exchanging tuples with different lengths. Such errors can be detected at compile time by using appropriate type inference algorithms, thus enforcing some kind of typing discipline on the usage of channels [14, 21, 19].

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

Recently, a considerable effort has been devoted to the problem of introducing objects into these calculi [8, 20, 24] thus providing a formal framework for modeling concurrent object-oriented languages. TyCO [20, 22] 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 [22], 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.

Example. Consider the following TyCO template definition for a memory cell with write and read methods:

$$\begin{gathered}\mathit{Cell} = (\mathit{self}\ \mathit{value})\ \mathit{self} \rhd \{\mathsf{write} : (\mathit{newval})\ \mathit{Cell}(\mathit{self}\ \mathit{newval})\ \&\\ \mathsf{read} : (\mathit{replyto})\ \mathit{replyto} \lhd \mathsf{val} : \mathit{value}\ ,\ \mathit{Cell}(\mathit{self}\ \mathit{value})\}\end{gathered}$$

An instance of *Cell* would be $\mathit{Cell}(x\ 2)$ for some variable x. The value in the memory cell is the integer 2. Sending a message $x \lhd \mathsf{write} : 5$ we replace the current cell contents with 5:

$$x \lhd \mathsf{write} : 5\ ,\ x \rhd \{\mathsf{write} : \cdots\} \longrightarrow \mathit{Cell}(x\ 5)$$

On the other hand, if we want the value in the cell to be read and sent to destination r then we invoke method read as follows:

$$x \lhd \mathsf{read} : r\ ,\ x \rhd \{\mathsf{write} : \cdots\} \longrightarrow r \lhd \mathsf{val} : 2\ ,\ \mathit{Cell}(x\ 2)$$

The abstract machine evolves from the object calculus by defining a static semantics (a type inference 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 this paper is organized as follows: sections 2 and 3 describe the abstract machine, its abstract syntax and dynamic semantics; section 4 introduces the static semantics; section 5 describes some of the properties verified by the abstract machine and, finally; section 6 compares our approach to other related work.

2 Abstract Syntax

Sequences and finite maps

1. $\vec{e}$ denotes the sequence $e_1 \ldots e_k$ $(k \geq 0)$ of elements of some syntatic 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 $f \upharpoonright V$, pronounced f *restricted to* V, is the finite map with domain $\mathrm{dom}(f) \cap V$ and values

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

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

Syntax

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

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

Syntatic 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 class statement is written as $X_1 = C_1$ and ... 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.

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

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:

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

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.

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

Comment. Packing variable bindings when making a closure for a thread or a set of methods, or dereferencing 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:

$$
\begin{array}{ll}
\textsc{Sched} & T, \_, Q, \mathsf{nil}, (B, \vec{P}) :: R \longrightarrow T, B, Q, \vec{P}, R \\[6pt]
\textsc{Scop} & \dfrac{a \notin \mathrm{dom}(Q)}{T, B, Q, |x|\, \vec{P} \mid \vec{P'}, R \longrightarrow T, B + \{x : a\}, Q + \{a : \bullet\}, \vec{P} \mid \vec{P'}, R} \\[6pt]
\textsc{Def} & T, B, Q, \mathsf{def}\ D\ \mathsf{in}\ \vec{P} \mid \vec{P'}, R \longrightarrow T + \{X : (B, C) \mid X = C \in D\}, B, Q, \vec{P} \mid \vec{P'}, R \\[6pt]
\textsc{RedMsg} & \dfrac{B(y) = a \qquad Q(a) = (B', M) :: Ms \qquad M(l) = (\vec{x})\vec{P'}}{T, B, Q, y \lhd l : [\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : Ms\}, \vec{P}, R :: (B' + \{\vec{x} : B(\vec{u})\}, \vec{P'})} \\[6pt]
\textsc{RedObj} & \dfrac{B(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{x})\vec{P'}}{T, B, Q, y \rhd M|\vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B + \{\vec{x} : \vec{v}\}, \vec{P'})} \\[6pt]
\textsc{RedInst} & \dfrac{T(X) = (B', (\vec{x})y \rhd M) \qquad B''(y) = a \qquad Q(a) = (l : [\vec{v}]) :: ms \qquad M(l) = (\vec{z})\vec{P'}}{T, B, Q, X[\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : ms\}, \vec{P}, R :: (B'' + \{\vec{z} : \vec{v}\}, \vec{P'})} \\[6pt]
\textsc{QueueMsg} & \dfrac{B(y) = a \qquad Q(a) = ms}{T, B, Q, y \lhd l : [\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : ms :: l : [B(\vec{u})]\}, \vec{P}, R} \\[6pt]
\textsc{QueueObj} & \dfrac{B(y) = a \qquad Q(a) = Ms}{T, B, Q, y \rhd M\ |\vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B, M)\}, \vec{P}, R} \\[6pt]
\textsc{QueueInst} & \dfrac{T(X) = (B', (\vec{x})y \rhd M) \qquad B'' = B' + \{\vec{x} : B(\vec{u})\} \qquad B''(y) = a \qquad Q(a) = Ms}{T, B, Q, X[\vec{u}]\ |\vec{P}, R \longrightarrow T, B, Q + \{a : Ms :: (B'', M)\}, \vec{P}, R}
\end{array}
$$

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 [20] 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:

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

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 $\mathsf{if}\ldots$ as a built-in both for performance and for keeping the design coherent with mappings of higher order constructs such as pattern matching [20]. The reduction rules for $\mathsf{if}\ c\ \mathsf{then}\ \vec{P}\ \mathsf{else}\ \vec{Q}$ (where c is a boolean constant) are the following:

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

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

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

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:

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

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 $\forall$ [5]. The *set of polymorphic types* is given by the following grammar:

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

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 Γ.

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).

5 Some Properties of the Abstract Machine

We first overview the semantics for TyCO .

5.1 Operational Semantics of TyCO

Scope restriction $|\,x\,|\,P$ and abstractions $(\vec{x})P$ are the *name binding operators* in the calculus, binding the free occurrences of x and $\vec{x}$ in the respective bodies P. 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 $\vec{v}$ for the free occurrences of names $\vec{x}$ in a process P, notation $P\{^{\vec{v}}/_{\vec{x}}\}$, defined only if the lengths of $\vec{v}$ and $\vec{x}$ match, and the names in $\vec{x}$ are pairwise distinct.

Declarations $\mathsf{def}\ X_1 = A_1\ \mathsf{and} \ldots \mathsf{and}\ X_n = A_n\ \mathsf{in}\ P$ constitute the only *variable binding operator* binding each variable X_i in each abstractions A_j, as well as in process P. The *set of free abstraction variables* in a process P, notation $\mathrm{fv}(P)$ is defined accordingly. The *domain* of an abstraction binding D, notation $\mathrm{dom}(D)$, is the set $\{X_1, \ldots, X_n\}$ when D is of the form $X_1 = A_1\ \mathsf{and} \ldots \mathsf{and}\ X_n = A_n$.

Alpha conversion is defined in the usual way, both for names and for abstraction variables. *Structural congruence* over processes simplifies the treatment of reduction. Following Milner [13], we define $\equiv$ to be the smallest congruence relation over processes generated by the following rules.

1. $P \equiv Q$ if P is α-convertible to Q,
2. $P \mid Q \equiv Q \mid P$, and $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, and $\mathbf{0} \mid P \equiv P$,
3. $|x|\mathbf{0} \equiv \mathbf{0}$, $|xy|P \equiv |yx|P$, and $|x|P, Q \equiv |x|(P, Q)$ if $x \notin \mathrm{fn}(Q)$,
4. $\mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0}$, $(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv\ \mathsf{def}\ D\ \mathsf{in}\ P \mid P'$ if $\mathrm{dom}(D) \cap \mathrm{fv}(Q) = \emptyset$,
5. $\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv\ \mathsf{def}\ D\ \mathsf{and}\ D'\ \mathsf{in}\ P$ if $\mathrm{dom}(D) \cap \mathrm{dom}(D') = \emptyset$,
6. $\mathsf{def}\ X = (\vec{y})Q\ \mathsf{and}\ D\ \mathsf{in}\ \ |\vec{x}|(X[\vec{a}] \mid P) \equiv\ \mathsf{def}\ X = (\vec{y})Q\ \mathsf{and}\ D\ \mathsf{in}\ \ |\vec{x}|(Q[^{\vec{a}}/_{\vec{y}}] \mid P)$,
7. $M \equiv N$ if M is a permutation of N, and $D \equiv D'$ if D' is a permutation of D.

One-step reduction, notation $P \to Q$, is the smallest relation generated by the following rules.

$$(\textsc{Com})\ a \lhd l[\vec{v}] \mid a \rhd M \to P[^{\vec{v}}/_{\vec{x}}] \qquad \text{where } M(l) = (\vec{x})P$$

$$(\textsc{Par})\ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad (\textsc{Res})\ \frac{P \to P'}{|x|P \to |x|P} \qquad (\textsc{Def})\ \frac{P \to P'}{\mathsf{def}\ D\ \mathsf{in}\ P \to \mathsf{def}\ D\ \mathsf{in}\ P}$$

$$(\textsc{Struct})\ \frac{P' \equiv P \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}$$

Multi-step reduction, or simply *reduction*, notation $P \twoheadrightarrow Q$, is the relation $\equiv \cup \to^{+}$, where $\to^{+}$ is the transitive closure of $\to$.

5.2 Translating Machine States into TyCO processes

In this section we encode the abstract machine states into equivalent TyCO processes.

The *Queue* associated with name a is encoded as sequences of messages, objects or just the nil process.

$$\begin{aligned}
[\![a : \bullet]\!] &\stackrel{def}{=} \mathsf{nil} \\
[\![a : ms :: m]\!] &\stackrel{def}{=} [\![a : ms]\!], a \lhd m \\
[\![a : Ms :: (B, M) Ms]\!] &\stackrel{def}{=} [\![a : Ms]\!], a \rhd M B
\end{aligned}$$

The encoding $[\![Q]\!]$ being the encodings for the queues put in parallel.

$$[\![\{a_1 : q_1, \cdots, a_n : q_n\}]\!] \stackrel{def}{=} [\![a_1 : q_1]\!] \mid \cdots \mid [\![a_n : q_n]\!]$$

The encoding for *Class Bindings*, $[\![T]\!]$, is:

$$[\![\{X_1 : (B_1, C_1), \cdots, X_n : (B_n, C_n)\}]\!] \stackrel{def}{=} X_1 = C_1 B_1 \,\mathsf{and} \cdots X_n : C_n B_n$$

The encoding $[\![R]\!]$ for the *Run-Queue* is:

$$[\![(B_1, \vec{P}_1), \cdots, X_n : (B_n, \vec{P}_n)\}]\!] \stackrel{def}{=} \vec{P}_1 B_1 \mid \cdots \mid \vec{P}_n B_n$$

Finally the encoding for a machine state, $[\![T, B, Q, \vec{P}, R]\!] = [\![S]\!]$ is:

$$[\![T, B, Q, \vec{P}, R]\!] \stackrel{def}{=} (\nu \vec{a})(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \vec{P}B, [\![Q]\!], [\![R]\!])$$

where $\{\vec{a}\} = dom(Q)$.

5.3 Properties

We now prove some interesting properties for the abstract machine.

Soundness: every state transition in the abstract machine can be viewed as a reduction or congruence between their process encodings in TyCO .

Theorem 1 (Soundness) *For S and S' abstract machine states, if $S \longrightarrow S'$ then $[\![S]\!] \equiv [\![S']\!]$ or $[\![S]\!] \longrightarrow [\![S']\!]$.*

Proof: enumerate all TyCOAM rules and check the validity of the transitions.
(Sched) we show that:

$$[\![T, B, Q, \mathsf{nil}, (B', \vec{P}) :: R]\!] \equiv [\![T, B', Q, \vec{P}, R]\!]$$

We know that,

$$\begin{gathered} [\![T, B, Q, \mathsf{nil}, (B', \vec{P}) :: R]\!] \stackrel{\text{def}}{=} \\ (\nu \vec{a})(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \mathsf{nil}B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\ [\![T, B', Q, \vec{P}, R]\!] \end{gathered}$$

since, $\mathsf{nil}B = \mathsf{nil}$ and $\mathsf{nil} \mid P \equiv P$ for any P.

(Scop) we show that:

$$[\![T, B, Q, (\nu x)\vec{P} \mid \vec{P'}, R]\!] \equiv [\![T, B + \{x : a\}, Q + \{a : \bullet\}, \vec{P} \mid \vec{P'}, R]\!] \qquad a \textit{ fresh}$$

We know that:

$$\begin{gathered} [\![T, B, Q, (\nu x)\vec{P} \mid \vec{P'}, R]\!] \stackrel{\text{def}}{=} \\ (\nu \vec{a})(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ ((\nu x)\vec{P} \mid \vec{P'})B \mid [\![Q]\!] \mid [\![R]\!]) \equiv_\alpha \\ (\nu \vec{a})(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ ((\nu a)\vec{P}[^a/_x] \mid \vec{P'})B \mid [\![Q]\!] \mid [\![R]\!]) \equiv \\ (\nu \vec{a}a)(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (\vec{P}[^a/_x] \mid \vec{P'})B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \end{gathered}$$

since $|\, x \,|\, P|Q \equiv |\, x \,|\, (P|Q)$ if $x \notin \mathrm{fn}(Q)$.

$$|\,\vec{a}a\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (\vec{P} \mid \vec{P}')(B + \{x : a\}) \mid [\![Q]\!] \mid [\![R]\!]) \equiv$$

since $a \notin \mathrm{fn}\, P'$

$$|\,\vec{a}a\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (\vec{P} \mid \vec{P}')(B + \{x : a\}) \mid \mathsf{nil} \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=}$$

since $\mathsf{nil} \mid P \equiv P$, and finally

$$[\![T, B + \{x : a\}, Q + \{a : \bullet\}, \vec{P} \mid \vec{P}', R]\!]$$

since $[\![a : \bullet]\!] \stackrel{\text{def}}{=} \mathsf{nil}$.

(Def) we show that:

$$[\![T, B, Q, \mathsf{def}\ D\ \mathsf{in}\ \vec{P}|\vec{P}', R]\!] \longrightarrow [\![T + T', B, Q, \vec{P} \mid \vec{P}', R]\!]$$

where $T' = \{X : (B, C) \mid X = C \in D\}$. We know that:

$$\begin{gathered}
[\![T, B, Q, \mathsf{def}\ D\ \mathsf{in}\ \vec{P}|\vec{P}', R]\!] \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (\mathsf{def}\ D\ \mathsf{in}\ \vec{P} \mid \vec{P}')B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (\mathsf{def}\ DB\ \mathsf{in}\ \vec{P}B) \mid \vec{P}'B \mid [\![Q]\!] \mid [\![R]\!])
\end{gathered}$$

Since $dom(D) \cap \mathrm{fv}(Q, R) = \emptyset$, when rename the processes obtaining:

$$\begin{gathered}
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \mathsf{def}\ DB\ \mathsf{in}\ \vec{P}B \mid \vec{P}'B \mid [\![Q]\!] \mid [\![R]\!]) \equiv \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{and}\ DB\ \mathsf{in}\ \vec{P}B \mid \vec{P}'B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{and}\ [\![T']\!]\ \mathsf{in}\ \vec{P}B \mid \vec{P}'B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{and}\ [\![T']\!]\ \mathsf{in}\ (\vec{P} \mid \vec{P}')B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
[\![T + T', B, Q, \vec{P} \mid \vec{P}', R]\!]
\end{gathered}$$

(RedMsg) we show that:

$$[\![T, B, Q, u \lhd l : [\vec{u}]|\vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : Ms\}, \vec{P}, R :: (B' + \{\vec{x} : B(\vec{u})\}, \vec{P}')]\!]$$

where $B(u) = a$, $Q(a) = (B', M) :: Ms$, and $M(l) = (\vec{x})\vec{P}'$. We know that:

$$\begin{gathered}
[\![T, B, Q, u \lhd l : [\vec{u}]|\vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (u \lhd l : [\vec{u}] \mid \vec{P})B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ a \lhd l : [B(\vec{u})] \mid \vec{P}B \mid [\![Q + \{a : Ms\}]\!] \mid a \rhd MB' \mid [\![R]\!]) \longrightarrow \\
|\,\vec{a}\,|\,(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \vec{P}B \mid [\![Q + \{a : Ms\}]\!] \mid ((MB')(l))[B(\vec{u})] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
\cdots \mid ((\vec{x})\vec{P}')B'[B(\vec{u})] \mid \cdots \stackrel{\text{def}}{=} \\
\cdots \mid ((\vec{x})(\vec{P}'(B - \{\vec{x}\})))[B(\vec{u})] \mid \cdots \equiv \\
\cdots \mid \vec{P}'(B - \{\vec{x}\})[^{B(\vec{u})}/_{\vec{x}}] \mid \cdots \stackrel{\text{def}}{=} \\
\cdots \mid \vec{P}'(B' + \{\vec{x} : B(\vec{u})\}) \mid \cdots \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : Ms\}, \vec{P}, R :: (B' + \{\vec{x} : B(\vec{u})\}, \vec{P}')]\!]
\end{gathered}$$

(RedObj) we show that:

$$[\![T, B, Q, y \triangleright M \mid \vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : ms\}, \vec{P}, R :: (B + \{\vec{x} : B(\vec{u})\}, \vec{P}')]\!]$$

where $B(y) = a$, $Q(a) = (l : [\vec{u}]) :: ms$, and $M(l) = (\vec{x})\vec{P}'$. We know that:

$$\begin{gathered}
[\![T, B, Q, y \triangleright M \mid \vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (y \triangleright M \mid \vec{P})B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ a \triangleright MB \mid \vec{P}B \mid [\![Q + \{a : ms\}]\!] \mid a \triangleleft l : [\vec{u}] \mid [\![R]\!]) \longrightarrow \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \vec{P}B \mid [\![Q + \{a : ms\}]\!] \mid M(l)[\vec{u}] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
\cdots\ ((\vec{x})\vec{P}')B[B(\vec{u})] \mid\ \cdots \stackrel{\text{def}}{=} \\
\cdots\ ((\vec{x})(\vec{P}'(B - \{\vec{x}\})))[B(\vec{u})] \mid\ \cdots \equiv \\
\cdots \mid \vec{P}'(B - \{\vec{x}\})[^{B(\vec{u})}/_{\vec{x}}] \mid\ \cdots \stackrel{\text{def}}{=} \\
\cdots \mid \vec{P}'(B' + \{\vec{x} : B(\vec{u})\}) \mid\ \cdots \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : ms\}, \vec{P}, R :: (B + \{\vec{x} : B(\vec{u})\}, \vec{P}')]\!]
\end{gathered}$$

(RedInst) we show that:

$$[\![T, B, Q, X[\vec{u}] \mid \vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : ms\}, \vec{P}, R :: (B'' + \{\vec{z} : \vec{v}\}, \vec{P}')]\!]$$

where $T(X) = (B', (\vec{x})y \triangleright M)$, $B''(y) = a$, $Q(a) = (l : [\vec{v}]) :: ms$, $M(l) = (\vec{z})\vec{P}'$ and $B'' = B' + \{\vec{x} : B(\vec{u})\}$. We know that:

$$\begin{gathered}
[\![T, B, Q, X[\vec{u}] \mid \vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ X[\vec{u}] \mid \vec{P}B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ X[B(\vec{u})] \mid \vec{P}B \mid [\![Q]\!] \mid [\![R]\!]) \equiv \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ [\![T]\!][B(\vec{u})] \mid \cdots) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (((\vec{x})y \triangleright M)B')[B(\vec{u})] \mid \cdots) \equiv \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ ((y \triangleright M)(B' - \{\vec{x}\}))[^{B(\vec{u})}/_{\vec{x}}] \mid \cdots) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (y \triangleright M)(B' + \{\vec{x} : B(\vec{u})\}) \mid \cdots) \stackrel{\text{def}}{=} \\
\cdots \mathsf{in}\ a \triangleright MB'' \mid \vec{P}B \mid [\![Q + \{a : ms\}]\!] \mid a \triangleleft l : [\vec{v}] \mid [\![R]\!]) \equiv\longrightarrow\equiv \\
\cdots \mathsf{in}\ \vec{P}B \mid [\![Q + \{a : ms\}]\!] \mid [\![R]\!] \mid MB''(l)[\vec{v}]) \stackrel{\text{def}}{=}\equiv\stackrel{\text{def}}{=} \\
\cdots \mid \vec{P}'(B'' + \{\vec{z} : \vec{v}\}) \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : ms\}, \vec{P}, R :: (B'' + \{\vec{z} : \vec{v}\}, \vec{P}')]\!]
\end{gathered}$$

(QueueMsg) we show that:

$$[\![T, B, Q, u \triangleleft l : [\vec{u}] | \vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : ms :: (l : [B(\vec{u})])\}, \vec{P}, R]\!]$$

where $B(u) = a$ and $Q(a) = ms$. We know that:

$$\begin{gathered}
[\![T, B, Q, u \lhd l : [\vec{u}] | \vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (u \lhd l : [\vec{u}] \mid \vec{P})B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
\cdots \mathsf{in} \mid a \lhd l : [B(\vec{u})] \mid \vec{P}B \mid \cdots\cdots \equiv \\
\cdots \mathsf{in}\ \vec{P}B \mid [\![Q]\!] \mid a \lhd l : [B(\vec{u})] \mid \cdots \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : ms :: (l : [B(\vec{u})])\}, \vec{P}, R]\!]
\end{gathered}$$

(QueueObj) we show that:

$$[\![T, B, Q, y \rhd M \mid \vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : Ms :: (B, M)\}, \vec{P}, R]\!]$$

where $B(y) = a$ and $Q(a) = Ms$. We know that:

$$\begin{gathered}
[\![T, B, Q, y \rhd M \mid \vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (y \rhd M \mid \vec{P})B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
\cdots \mathsf{in} \mid a \rhd MB \mid \vec{P}B \mid \cdots\cdots \equiv \\
\cdots \mathsf{in}\ \vec{P}B \mid [\![Q]\!] \mid a \rhd MB \mid \cdots \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : Ms :: (B, M)\}, \vec{P}, R]\!]
\end{gathered}$$

(QueueInst) we show that:

$$[\![T, B, Q, X[\vec{u}] \mid \vec{P}, R]\!] \longrightarrow [\![T, B, Q + \{a : Ms :: (B'', M)\}, \vec{P}, R]\!]$$

where $T(X) = (B', (\vec{x})y \rhd M)$, $B''(y) = a$, $Q(a) = Ms$ and $B'' = B' + \{\vec{x} : B(\vec{u})\}$. We know that:

$$\begin{gathered}
[\![T, B, Q, X[\vec{u}] \mid \vec{P}, R]\!] \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ X[\vec{u}] \mid \vec{P}B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ X[B(\vec{u})] \mid \vec{P}B \mid [\![Q]\!] \mid [\![R]\!]) \equiv \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ [\![T]\!][B(\vec{u})] \mid \cdots) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (((\vec{x})y \rhd M)B')[B(\vec{u})] \mid \cdots) \equiv \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ ((y \rhd M)(B' - \{\vec{x}\}))[{}^{B(\vec{u})}/_{\vec{x}}] \mid \cdots) \stackrel{\text{def}}{=} \\
|\vec{a}|(\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ (y \rhd M)(B' + \{\vec{x} : B(\vec{u})\}) \mid \cdots) \stackrel{\text{def}}{=} \\
\cdots \mathsf{in}\ a \rhd MB'' \mid \vec{P}B \mid [\![Q]\!] \mid [\![R]\!]) \stackrel{\text{def}}{=} \\
[\![T, B, Q + \{a : Ms :: (B'', M)\}, \vec{P}, R]\!] \ \square
\end{gathered}$$

Completeness The abstract machine is clearly *not* complete relative to TyCO . This essentially results from two restrictions we imposed on the behavior of processes. The first such restriction is easily demonstrated with the following example in TyCO :

$$a \lhd l : [\vec{v}] \mid a \rhd \{l : (\vec{x})P\} \mid a \rhd \{l : (\vec{x})Q\}$$

which may reduce as:

$$\{{}^{\vec{v}}/_{\vec{x}}\}P \mid a \rhd \{l : (\vec{x})Q\} \qquad or \qquad \{{}^{\vec{v}}/_{\vec{x}}\}Q \mid a \rhd \{l : (\vec{x})P\}$$

The corresponding sequence in the abstract machine has the parallel composition operator replaced by a concatenation operator (with the same syntax, though). Parallel compositions of atomic processes are transformed into strings of processes. The abstract machine goes through these strings sequentially, without any kind of re-ordering. Thus the only possible outcome of the above example when it is run by the machine is equivalent to the composition:

$$\{^{\vec{v}}/_{\vec{x}}\}P \mid a \triangleright \{l : (\vec{x})Q\}$$

Forcing sequential execution of processes eliminates non-determinism within threads in the abstract machine. Also, the use of a run-queue to hold newly generated threads serializes their execution. Non-determinism is lost in a serial implementation of the abstract machine. However in a concurrent implementation we could eliminate the SCHED rule and introduce non-determinism in the computation due to thread interactions (on shared names).

The second important restriction is the fact that we close processes for names. This restricts their behavior in one fundamental: abstract machine programs are closed entities uncapable to react or interface with external object systems.

At this point it is not clear what the relation is (if any) between the semantics of TyCO and that of the abstract machine.

Communication/Method-closure Queue Invariant: the abstract machine preserves the invariant that at any time during a computation the queues associated with names have either communications or method-closures (or empty). Formally, we have the following lemma:

Lemma 1 *Denote S_0 as the initial state of the abstract machine. Then, if $S_0 \twoheadrightarrow \_, \_, Q, \_, \_$ then* $\forall_{a \in \mathrm{dom}(Q)} Q(a) \in \mathrm{QC{\small OMM}}$ *or* $Q(a) \in \mathrm{QM{\small ETH}}$.

Proof Outline: Induction on the length of the reduction performed by the abstract machine.

We start with the initial state of the abstract machine:

$$S_0 = \emptyset, \emptyset, \emptyset, \vec{P}, \bullet$$

where $\vec{P}$ is the initial thread. There are no queues in this state. The invariant holds trivially.

Now assume that we are in a generic state: T, B, Q, P, R for which the invariant holds. We must check that after applying any of the reduction rules to such a state the invariant will still holds.

Rules SCHED and DEF do not affect the queues in Q, so after an application the invariant will still hold. Also the SCOP rule creates a new name and an empty queue with it in Q. Since the empty queue is both in QComm and QMeth the invariant is preserved.

The RED and QUEUE rules affect the queues in a more complex way. We enumerate the possibilities:

- REDMSG is used whenever the queue being inspected is in QMeth . It removes one of the method-closures from the queue. The remainder of the queue (eventually empty) is still in QMeth ;
- REDOBJ (same as above but with the queue in QComm);
- REDINST (same as above but with the queue in QComm);

The RED rules preserve the invariant. Similarly, for the QUEUE rules we have:

- QUEUEMSG is used whenever the queue being inspected is in QComm . It adds another communication to the end of the queue which obviously remains in QComm ;

- QueueObj (same as above but with the queue in QMeth);
- QueueInst (same as above but with the queue in QMeth);

The Queue rules also preserve the invariant.□

Absence of Deadlocks: next we prove that the abstract machine does not deadlock for well typed programs.

Definition 1 *S is a deadlock state if $S \neq \_, \_, \_, \mathsf{nil}, \bullet$ and $S \not\rightarrow$.*

Definition 1 *Let S_0 be the initial state of the machine. S_0 deadlocks if $S_0 \twoheadrightarrow S$ and S is a deadlock state.*

Theorem 1 *If $\vec{P}$ in $S_0 = \emptyset, \emptyset, \emptyset, \vec{P}, \bullet$ is well-typed then S_0 does not deadlock.*

Proof outline: we know that $S_0 \twoheadrightarrow T, B, Q, \vec{P}', R$. We analyse the structure of thread $\vec{P}'$.

If it is the empty thread: nil, we have two possibilities:

- $R = \bullet$. In this case we have a state of the form $\_, \_, \_, \mathsf{nil}, \bullet$ which by definition is not a deadlock state.
- $R \neq \bullet$. In this case we apply the Sched rule to initiate the execution of a new thread with the corresponding bindings.

Now assume that our thread has the form: $P \mid \vec{P}'$ and show that we may always apply one of the reduction rules successfully. We do this by enumeration of the processes in the abstract machine:

- $P = \mid x \mid \vec{P}'$. Since all bound variables in the initial process are distinct (variable renaming assumption) we apply the Scop rule.
- $P = \mathsf{def}\ D\ \mathsf{in}\ \vec{P}$. Again, the variable renaming assumption implies that $\mathrm{dom}(T) \cap \mathrm{dom}(D) = \emptyset$. We apply the Def rule.
- $P = y \lhd l : [\vec{u}]$. First, $y \in \mathrm{dom}(B)$ (processes are closed) and $a = B(y) \in \mathrm{dom}(Q)$ (processes are closed and well typed). Two situations may now occur: $Q(a) \in$ QComm or $Q(a) \in$ QMeth . This results directly from the queue invariant. If $Q(a) \in$ QComm we apply the QueueMsg since $\{\vec{u}\} \subseteq \mathrm{dom}(B)$. Otherwise, if $Q(a) \in$ QMeth then we have an opportunity to reduce and apply the RedMsg rule. This is possible since programs are well typed and hence, the method-closure taken from the queue contains a method $l : (\vec{x})\vec{P}$ such that $|\vec{x}| = |\vec{u}|$.
- $P = y \rhd M$. First, $y \in \mathrm{dom}(B)$ (processes are closed) and $a = B(y) \in \mathrm{dom}(Q)$ (processes are closed and well typed). Again the queue invariant ensures that only two situations may occur: $Q(a) \in$ QComm or $Q(a) \in$ QMeth . If $Q(a) \in$ QMeth we apply the QueueObj with no further checking. If, on the other hand, $Q(a) \in$ QComm then we apply the RedObj rule. This is possible since programs are well typed and hence, the current object contains a method $l : (\vec{x})\vec{P}$ such that $|\vec{x}| = |\vec{u}|$.
- $P = X[\vec{u}]$. This is essentially the same as the above, with the additional remark that, since programs are closed $X \in \mathrm{dom}(T)$.

From the above we conclude that no well typed program deadlocks the abstract machine.□

6 Related Work

Efforts to provide a framework for concurrent object-oriented languages have relied on two fundamental approaches: the concept of *actors* [3, 4] with the associated computational model, and; encodings of objects into Milner, Parrow and Walker's π-calculus or an equivalent asynchronous formulation due to Honda and Tokoro [8].

Some actor languages have been implemented with some success such as the ABCL family [26, 25]. Moreover, most existing concurrent object-oriented languages incorporate aspects of the actors model in their specifications.

In recent years researchers have devoted a lot of effort in providing semantics for objects within the asynchronous π-calculus [8, 24, 16, 12], variations [20, 6] and extensions [7] (for constraint programming). Although there is extensive theoretical research on the subject, abstract machine specifications and implementations of actual models are still scarce [19, 18, 10, 11]. A few programming languages based on process calculi have been purposed. Some, such as Pict [19] or Oz [10, 11] have gained some prominence.

The TyCO object calculus on which the abstract machine described in this paper is based is very reminescent of Abadi and Cardelli's ς-calculus [1, 2]. Here as in TyCO objects are primitive and represented as sums of labelled methods. Labelled method invocation (messages in TyCO) are also present. The calculus has an extra method update operation which is not present in TyCO and is fully sequential. TyCO has the same computational power as the asynchronous π-calculus [22].

Pict is based on an asynchronous form of Milner's π-calculus, with the basic abstractions being processes and names (channels). Processes communicate atomically by sending values along shared channels. Recursion is modeled by process replication. On the other hand, Oz is based in the γ-calculus, where the basic abstractions are: names, variables (logical), procedural abstraction and cells. Both calculi are specializations of a more general calculus that is obtained from the polyadic π-calculus by distinguishing between names and variables and making the variables logical. Our abstract machine is based on TyCO [20], a form of the asynchonous π-calculus where messages (labeled output processes with a *nil* continuation) and objects (non replicated input processes in the form of collections of labeled methods) are first class abstractions. Replication of processes is modeled through instantiation of recursive classes (objects abstracted on a set of variables).

Pict is a pure process-based concurrent programming language. Besides a subset of the basic π-calculus functionality, Pict also defines some higher level abstractions such as templates and *records* (structured values with named fields) and provides a type-inference system. Objects are implemented using records. As a result method invocation in such objects is a three-way protocol (communication, matching,communication),as opposed to a one-way protocol (communication) in our abstract machine. This results from the fact that messages and objects are first-class entities in our design.

Another important property of objects is the fact that method selection is based on labeled transitions. Besides a faster invocation protocol this has the added benefit of not producing computational *garbage*. In Pict objects are encoded as records of names. Each *method* waits on one of these names for communication. The creation of an object implies the creation of one new name per method (and one for each state variable) and for the method processes to be placed in parallel, thus forcing the creation of closures. When a method invocation is received, the body of the method is executed with the substitutions performed. This produces a large amount of *garbage* in the form of unused process closures and names, which must be dealt with both in theory and in the context of a real implementation.

Oz combines three major programming paradigms: functional, object-oriented and constraint logic programming, in to an high-level programming language. Constraints are built over logical variables by first-order logic equations, eventually using a set of predefined predicates. Cells are primitive entities that maintain state and provide atomic read-write operations. Procedures have encapsulated state so that objects can be defined directly as sets of methods (procedures) acting over their state (represented as a set of logical variables). This representation is quite close to objects in our own abstract machine if we replace the state held in logical variables with the class formal parameters.

The Turner Abstract Machine (TAM [19, 18]), which is the basis for the implementation of Pict introduced some simplifications into the semantics for the asynchronous π-calculus to allow for an efficient implementation. First, only input processes were allowed to replicate. This is also true for our abstract machine since recursion can only occur within objects (instances of classes). Furthermore, the form of replicated input processes was changed so that replication occurred only after a communication on the channel was received. This property also holds in our abstract machine but does not result from any design restriction. An object, instance of some recursive class, can only be replicated when one of its methods is invoked. A recursive call within the body of the method may then create a new replica of the object (with possibly distinct argument values).

Turner realized that π-calculus processes were far too pervasive and short lived to allow an efficient implementation by mapping them one-to-one with threads. The Pict abstract machine tries to maximize the number of processes executed within a thread of control. This is achieved by allowing a process to be executed at once instead of making a closure for it and placing it in the run-queue. The only exceptions occur when an (replicated) input process closure is taken from one of the queues upon reception of a communication.

On the other hand the Oz abstract machine executes programs (closed for names) made up of threads: stacks of expressions. State (in the form of cells) and equations are kept in a *blackboard* which is updated by concurrent threads. New threads are created whenever an expression occurs in the current thread that cannot be imediately reduced. The expression is executed in the new thread. Thread composition is commutative.

In our design we introduced a new syntactic category for threads. They are sequences of processes linked by the "|" (*concatenation*) operator. The machine executes threads. All newly created threads are placed at the end of the run-queue (in a concurrent implementation they would be run in a new thread). Processes within threads are executed without interference from the scheduler, hence threads cannot be blocked. Threads are identified with methods from objects. This fact makes for a particularly clean model for an object-oriented language where objects are sets of methods atomically executed (with some *state* carried by class parameters) [9]. Moreover, in a concurrent setting, non-determinism originates in interactions of threads on shared names. Actually, *persistent* objects in our abstract machine are reminescent of actors [3, 4, 9]. They may initiate communications with other objects or even to themselves (by sending messages to *self*); they may create new objects either by prototyping or instantiation and propagate *self* channels and; they may change behavior after an incomming communication. This can be achieved by creating a new object at *self* with the same type (interface) but eventually with distinct method implementations.

The abstract machine implements the current formulation of TyCO [23] in its full extent. The Pict abstract machine, on the other hand, implements the asynchronous π-calculus with a few important restrictions and semantic nuances. First, the choice (*summation*) construct is not supported directly in the abstract machine. Rather it is provided by the implementation as an extra library module. This is both to allow an efficient run-time representation of names and avoid the costly garbage collection of unused names in choice patterns [17]. Also, the semantics for replication was restricted in such a way that (replication) can only occur in input processes after a communication is received. This is to allow the abstract machine to efficiently identify the places where process replication occurs [19].

An implementation of the abstract machine is underway with both the compiler and run-time system nearly completed. Future work will focus on performance evaluation and optimizing the system. Latter we will introduce higher level constructs into the abstract syntax, such as expressions (functional, pattern matching, algebraic types) and inheritance.

References

[1] M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.

[2] M. Abadi and L. Cardelli. A Theory Of Primitive Objects: Untyped and First-Order System. *Information and Computation*, 125(2):78–102, 1996.

[3] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. *Research Directions on Object-Oriented Programming*, 1981. Shiver and Wegner, editors. MIT Press.

[4] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

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

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

[7] 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.

[8] 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.

[9] D.G. Kafura and R.G. Lavender. Concurrent Object-Oriented Languages and the Inheritance Anomaly . In *International Summer Institute on Parallel Computer Architectures, Languages, and Algoritms (ISIPCALA93), Prague*, pages 183–213. Czech ACM Chapter, July 1993.

[10] G. Smolka M. Henz and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. pages 29–48, 1993.

[11] R. Scheidhauer M. Mehl and C. Schulte. An Abstract Machine for Oz. Technical report, German Research Center for Artificial Intelligence (DFKI), June 1995.

[12] 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.

[13] R. Milner. *Communication and Concurrency*. Preentice-Hall, 1989.

[14] R. Milner. The Polyadic π-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh , October 1991.

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

[16] 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.

[17] B. Pierce and D. Turner. Concurrent Objects in a Process Calculus.

[18] B. Pierce and D. Turner. The Pict Abstract Machine. June 1995. draft version.

[19] David Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

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

[21] V. Vasconcelos and K. 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.

[22] 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.

[23] Vasco T. Vasconcelos. Typed Concurrent Objects. to be published, June 1997.

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

[25] A. Yonezawa, E. Shibayama, and et. al. Modelling and Programming in an Object-Oriente d Concurrent Language ABCL/1. In M. Tokoro, editor, *Object-Oriented Programming*, pages 55–89. MIT Press, 1987.

[26] Akinori Yonezawa. *ABCL, an Object Oriented Concurrent System.* MIT Press, 1990.