Programming and Implementation Issues in Non-Uniform TyCO

António Ravara * Luís Lopes †

Abstract

We present an abstract machine specification for non-uniform TyCO. It is, up to our knowledge, the first proposal of an abstract machine to compile a process calculus of non-uniform objects.

TyCO is a name-passing calculus of concurrent objects. Since concurrent objects do not always offer the same methods due to synchronisation constraints, we advocate non-uniform interfaces, i.e., interfaces with only enabled methods. From the programming point of view the use of non-uniform interfaces for representing objects, as opposed to the uniform record interfaces, allows a more intuitive and expressive definition of objects and in general object-based data structures. This is a result of the far richer behaviours allowed by the non-uniformity. From an implementation point of view, frequently the code is much shorter, both in the source programs as well as in the final byte code or other executable format; even in cases where the difference in code size is not significant, the use of a single name for all of the object's dynamic instances substantially reduces heap space, and allows optimisations in heap usage.

The abstract machine for non-uniform TyCO grows from a similar specification for the uniform case. The changes are very localised and regard essentially the structure of individual communication channels. Based on experience gathered with an implementation of uniform TyCO, we discuss the advantages (and disadvantages) of non-uniformity from the point of view of performance.

1 Introduction

TyCO– Typed Concurrent Objects – is a form of the asynchronous π-calculus [4, 9] featuring first class objects as input-guarded labelled sums, asynchronous labelled messages as output selectors, and template processes [18]. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. Continuations implement synchronous communication. Templates are parameterised processes polymorphic on the parameter variables allowing, for example, to model classes. Explicit instantiation of recursive templates models unbounded behaviour.

As a calculus, TyCO is the base of the programming language Core-TyCO [19], for which there is an implementation [10]. A type system [18] assigns monomorphic types to variables and polymorphic types to template variables, that is, template processes are polymorphic on their parameters.

TyCO is reminiscent of the Abadi and Cardelli's ς-calculus [1] in the sense that objects are sums of labelled methods (the name *self* being interpreted as the channel where the object is located) and messages can be seen as asynchronous method invocations. The ς-calculus however presents a method update operation which is not present in TyCO.

*Department of Mathematics, Instituto Superior Técnico,e-mail: amar@math.ist.utl.pt

†Department of Computer Science, Universidade do Porto, e-mail: lblopes@ncc.up.pt

2 Non-Uniform TyCO

Concurrent objects can offer services non-uniformly, according to synchronisation constraints, i.e., the availability of a service depends on the internal state of the object (which reflects the state of the system) [14]. Objects with methods enabled or disabled according to its internal state are very common in object-oriented programming (e.g. a stack, a finite buffer, an ftp server, a bank account, a cash machine). Whence, we aim at a more flexible style of programming and at a type discipline that copes with non-uniform concurrent objects: the interfaces of the objects only have enabled methods, and when a client asks for a disabled method the message is not rejected if the internal state of the object may change such that the method becomes available. A pair object-message – a redex – is not an error if the message may be accepted at some time in the future. Usually a redex that does not reduce is considered an error. To achieve the objectives previously referred, we provide a looser definition of communication error. We call bad redexes those that do not reduce, and we do not consider them as errors if the object may evolve to a state where it can accept the message. A communication error is a process with a persistent bad redex, either because the object is deadlocked, or because it will never have the right method for the message.

Traditional type systems assign rigid interface-like types to the names of the objects [6, 13, 18]. Type-systems that support objects with non-uniform interfaces, based on the distinct definition of a run-time protocol error discussed above, have been proposed for TyCO [15, 16]. The systems ensure safety properties resulting from the absence of persistent bad redexes, namely the absence of some deadlocks (that we call local), and the guarantee that typable processes will not run into communication errors.

Take the typical example of a one-place buffer that only allows read/consume operations when it is full, and write/produce operations when it is empty. We like to specify it as follows, since it shows clearly the behaviour of the buffer, alternating between write and read operations.

```
def Empty(b) = b?{ write(u) = Full[b,u] }
and Full(b,u) = b?{ read(r) = r!val[u] | Empty[b] }
in inaction
```

This style of programming is state-dependent, that is, for each object we specify its different interfaces according to the availability of services. The example presented can easily be scaled up to an n-place finite buffer by introducing a third state, where the buffer is neither empty nor full. Objects usually have a finite (and small) number of such states, thus it is not difficult to program in this style. The referred type systems reject the example above, since name b do not have a single interface containing both labels *write* and *read*. An alternative typable implementation, uses the busy-waiting technique to handle non-available operations.

```
def Buf(b,v,empty) =
      b?{ read(r)  = if empty then b!read[r] | Buf[b,v,true]
                     else r![v] | Buf[b,v,true],
          write(u) = if empty then Buf[b,u,false]
                     else b!write[u] | Buf[b,v,false] }
in inaction
```

In the second implementation, a process containing the redex $\mathrm{Buf}[b, v, \mathit{true}] \mid b!\mathit{read}[r]$ is not an error, and the presence of a message of the form $b!\mathit{write}[u]$ makes possible the acceptance of the *read* message. Similarly, in the first implementation a process containing the redex $\mathrm{Empty}[b] \mid b!\mathit{read}[r]$ should not be considered an error, as again, the presence of a message like $b!\mathit{write}[u]$ also makes the reception of the *read* message possible. Nonetheless, notice that a process locally deadlocked in name b, like **new** $b\ \mathrm{Empty}[b] \mid b!\mathit{read}[r]$ is an error. In conclusion, the objects behave similarly, but the first implementation is more informative and more natural regarding the buffer's behaviour.

Other interesting examples are an Automatic Teller Machine and a Pop3 Server. They show how to scale-up the previous example. The first is a typical non-uniform object, as the services provided depend on various synchronisation constraints: the status of the machine (online or off-line), the type of user's card, the kind of service the user requests, etc. We present them below in some detail, both with uniform and non-uniform code.

Example 2.1 (ATM)
Consider an implementation of an ATM with two parameters, its address and the bank's address. A request for a session with the ATM creates a new thread which establishes a private communication in b, offering recurrently a collection of methods: deposit, withdraw, and balance. Several users can concurrently use the ATM.

1. *Uniform implementation. After each request of a method the user has a new private address to communicate with, since the same name cannot have several interfaces. An alternative is to have a single interface with all methods, whether they are enabled or not. Moreover, reply messages to the user have always the same aspect, to ensure well-typedness.*

```
def Atm(a,b) = a?req(x) = new s x!rep[s] | Atm[a,b] | Action[s,b]

and Action(s,b) =
      s?{ deposit(x)  = new z x!rep[z] | Dep[z,s,b] ,
          withdraw(x) = new z x!rep[z] | Get[z,s,b] ,
          balance(x)  = new z x!rep[z] | State[z,s,b] ,
          quit(x)     = x!bye }

and Dep(z,s,b) =
      z?{ amount(u,x) = new v b!dep[u,v] |
                          v?{ done = x!ok[0] | Action[s,b] ,
                              prob = x!sorry | Action[s,b] } ,
          reset(x)    = x!ok[0] | Action[s,b] }

and Get(z,s,b) =
      z?{ five(x)     = new v b!get[5,v] |
                          v?{ done = x!ok[5] | Action[s,b] ,
                              prob = x!sorry | Action[s,b] } ,
          ten(x)      = new v b!get[10,v] |
                          v?{ done = x!ok[10] | Action[s,b] ,
                              prob = x!sorry | Action[s,b] } ,
          twenty(x)   = new v b!get[20,v] |
                          v?{ done = x!ok[20] | Action[s,b] ,
                              prob = x!sorry | Action[s,b] } ,
          amount(u,x) = new v b!dep[u,v] |
                          v?{ done = x!ok[u] | Action[s,b] ,
                              prob = x!sorry | Action[s,b] } ,
          reset(x)    = x!ok[0] | Action[s,b] }

and State(z,s,b) =
      z?val(x) = new v b!val[v] |
                 v?{ done(u) = x!ok[u] | Action[s,b] ,
                     prob    = x!sorry | Action[s,b] }

in ...
```

2. *Non-uniform implementation. One private address for communication with the user suffices. It is the address of the object that offer the ATM's services, and the interfaces are state dependent. Note that the reply messages do not always have the same aspect, and the code of some of the private object's states is simpler.*

```
def Atm(a,b) = a?req(x) = new s x!rep[s] | Atm[a,b] | Action[s,b,x]

and Action(s,b,x) =
      s?{ deposit  = Dep[s,b,x] ,
          withdraw = Get[s,b,x] ,
          balance  = State[s,b,x] ,
          quit     = x!bye }

and Dep(s,b,x) =
      s?{ amount(u) = new v b!dep[u,v] |
                      v?{ done = x!ok | Action[s,b,x] ,
                          prob = x!sorry | Action[s,b,x] } ,
          reset    = x!ok | Action[s,b,x] }

and Get(s,b,x) =
      s?{ five      = new v b!get[5,v] |
                      v?{ done = x!ok[5] | Action[s,b,x] ,
                          prob = x!sorry | Action[s,b,x] } ,
          ten       = new v b!get[10,v] |
                      v?{ done = x!ok[10] | Action[s,b,x] ,
                          prob = x!sorry | Action[s,b,x] } ,
          twenty    = new v b!get[20,v] |
                      v?{ done = x!ok[20] | Action[s,b,x] ,
                          prob = x!sorry | Action[s,b,x] } ,
          amount(u) = new v b!dep[u,v] |
                      v?{ done = x!ok[u] | Action[s,b,x] ,
                          prob = x!sorry | Action[s,b,x] } ,
          reset     = x!ok | Action[s,b,x] }

and State(s,b,x) = new v b!val[v] |
                   v?{ done(u) = x!ok[u] | Action[s,b,x] ,
                       prob    = x!sorry | Action[s,b,x] }

in ...
```

Example 2.2 (POP3 Server)
A rendez-vous with a Pop3 server triggers a new thread b for private communication with the client. The session starts after the client have passed the authorisation protocol: first sends its username, then a valid password.

1. *Uniform implementation. Each step of the protocol creates a new private name to communicate with the client, as the interfaces are always different.*

```
def Init(pid) = new nis Pop3d[pid]

and Pop3d(pid) = pid?rv(r) = new b r!ok[b] | Auth[b,r] | Pop3d[pid]

and Auth(b,r) =
      b?{ user(uid) = new s r!ok[s] | Auth1[s,r,uid] ,
          quit(r)   = Update[b,r] }

and Auth1(b,r,uid) =
      b?{ user(newuid) = new s r!ok[s] | Auth1[s,r,newuid] ,
          pass(pas)    = new w nis![w] |
```

```
                      w?ok(t) = new s t!chkusr[uid,pas,s] |
                                s?{ invalid = r!sorry,
                                    valid   = Trans[b,r] } ,
          quit(r)      = Update[b,r] }

and Trans(b,r) =
      b? { stat    = r!done | Trans[b,r] ,
           list(k) = r!done | Trans[b,r] ,
           retr(k) = r!done | Trans[b,r] ,
           rset    = r!done | Trans[b,r] ,
           del(k)  = r!done | Trans[b,r] ,
           noop    = r!done | Trans[b,r] ,
           quit    = Update[b,r] }

and Update(b,r) = -- update state after transactions |
                  r!bye

in ...
```

2. *Non-uniform implementation. There is a single thread for private communication with the client, although the interfaces of each state are different.*

```
def Init(pid) = new nis Pop3d[pid]

and Pop3d(pid) = pid?rv(r) = new b r!ok[b] | Auth[b,r] | Pop3d[pid]

and  Auth(b,r) = b?{ user(uid) = Auth1[b,r,uid] ,
                     quit(r)   = Update[b,r] }

and  Auth1(b,r,uid) =
       b?{ user(newuid) = Auth1[b,r,newuid] ,
           pass(pas)    = new w nis![w] |
                          w?ok(t) = new s t!chkusr[uid,pas,s] |
                                    s?{ invalid = r!sorry,
                                        valid   = Trans[b,r] } ,
           quit(r)      = Update[b,r] }

and Trans(b,r) =
      b? { stat    = r!done | Trans[b,r] ,
           list(k) = r!done | Trans[b,r] ,
           retr(k) = r!done | Trans[b,r] ,
           rset    = r!done | Trans[b,r] ,
           del(k)  = r!done | Trans[b,r] ,
           noop    = r!done | Trans[b,r] ,
           quit    = Update[b,r] }

and Update(b,r) = r!bye

in ...
```

3 Typed Concurrent Objects

Syntax. In the sequel we describe the syntactic categories required in the definition of the TyCO calculus. *Names* are locations where communication occurs and are the only basic type of the calculus. *Variables* stand for names in a given context. They are introduced syntactically in template definitions, method parameters and creation of new names. They are bound to names when a reduction occurs or when a new name is created. *Labels* identify methods in objects and are not first class. *Template Variables* identify process templates and are also not first class entities.

NOTATION 3.1 *Let us fix some terminology.*

1. *Syntactic categories.*

$$\begin{array}{rll} a \in & \mathit{Name} & \qquad\qquad \mathit{Names} \\ x, y \in & \mathit{Var} & \qquad\qquad \mathit{Variables} \\ u, v \in & \mathit{Var} \cup \mathit{Name} & \qquad\qquad \mathit{Values} \\ l \in & \mathit{Label} & \qquad\qquad \mathit{Labels} \\ X \in & \mathit{TVar} & \qquad\qquad \mathit{Template\ variables} \end{array}$$

2. *A* method collection *is a map that associates a label and a set of variables with a process – the body of the method. Similarly,* template definition *is a map associating a template variable and a set of variables with a process – the body of the definition.*

$$\begin{array}{ll} M \in \mathit{Meth} = \mathit{Label} \times \mathit{Var}^* \mapsto \mathit{Proc} & \qquad\qquad \mathit{Methods} \\ D \in \mathit{DBind} = \mathit{TVar} \times \mathit{Var}^* \mapsto \mathit{Proc} & \qquad\qquad \mathit{Template\ Bindings} \end{array}$$

Let $\tilde{v}$ stand for a sequence of names, and $\tilde{x}$ stand for a sequence of pairwise distinct names. Moreover, let $\{\tilde{v}\}$ stand for the set of names in the sequence $\tilde{v}$.

DEFINITION 3.2 (PROCESSES) *The following grammar defines the set of TyCO processes Proc, ranged over by P.*

$$\begin{array}{rcll} P & ::= & \textsf{inaction} & \mathit{terminated\ process} \\ & \mid & a!l[\tilde{v}] & \mathit{asynchronous\ message} \\ & \mid & a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\} & \mathit{object} \\ & \mid & \textsf{new}\ x\ P & \mathit{variable\ declaration} \\ & \mid & P \mid P & \mathit{parallel\ composition} \\ & \mid & X[\tilde{v}] & \mathit{instance} \\ & \mid & \textsf{def}\ D\ \textsf{in}\ P & \mathit{recursion} \\ & \mid & (P) & \mathit{grouping} \\ M & ::= & \{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\} & \mathit{method\ collection} \\ D & ::= & X_1(\tilde{x}_1) = P_1\ \textsf{and} \ldots \textsf{and}\ X_n(\tilde{x}_n) = P_n & \mathit{template\ declaration} \end{array}$$

NOTATION 3.3 *We assume some syntactic conventions for processes in TyCO, namely:*

- *The labels l_i are pairwise distinct in a method collection $\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$;*
- *The template variables X_i are pairwise distinct in a template definition* $X_1(\tilde{x}_1) = P_1$ **and** $\ldots$ **and** $X_n(\tilde{x}_n) = P_n$.

Binding Operators. Parenthesis and **new** constitute the binding operators for variables. Variables $\tilde{x}$ are bound in the part P of a method $l(\tilde{x}) = P$, a definition $X(\tilde{x}) = P$ or a scope restriction **new** $\tilde{x}$ P. We define accordingly the set of free variables of a process P, denoted $\mathrm{fv}(P)$, the set of *bound variables* and the substitution of variables by constants, denoted $\{\tilde{v}/\tilde{x}\}P$. A process P is *closed for variables* if $\mathrm{fv}(P) = \emptyset$.

Also, **def** is the binding operator for template variables, the template variable X being bound to Q in **def** $X(\tilde{x}) = P$ **in** Q. Accordingly, we define the set of free template variables of a process P, denoted $\mathrm{ft}(P)$, and the set of *bound template-variables.* A process P is *closed for template-variables* if $\mathrm{ft}(P) = \emptyset$.

Dynamic Semantics. TyCO's semantics specifies a set of rules that formally describe how processes reduce. Following Milner [12] we divide the semantics of the calculus in two parts: the *structural congruence* rules and the *reduction* rules.

Structural congruence rules allow the re-writing of processes into semantically equivalent expressions until they may be reduced using reduction rules.

Definition 3.4 (Structural Congruence) *We define $\equiv$ to be the smallest congruence relation over processes generated by the following rules.*

1. $P \equiv Q$ *if* $P \equiv_\alpha Q$
2. $P \mid \textsf{inaction} \equiv P, \qquad P \mid Q \equiv Q \mid P, \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
3. $\textsf{new}\ x\ \textsf{inaction} \equiv \textsf{inaction}$,
   $\textsf{new}\ x\ \textsf{new}\ y\ P \equiv \textsf{new}\ y\ \textsf{new}\ x\ P$,
   $\textsf{new}\ x\ P \mid Q \equiv \textsf{new}\ x\ (P \mid Q) \qquad$ *if* $x \notin \mathrm{fv}(Q)$
4. $\textsf{def}\ D\ \textsf{in}\ \textsf{inaction} \equiv \textsf{inaction}$,
   $\textsf{def}\ D\ \textsf{in}\ \textsf{new}\ x\ P \equiv \textsf{new}\ x\ \textsf{def}\ D\ \textsf{in}\ P \qquad$ *if* $x \notin \mathrm{fv}(\textsf{def}\ D\ \textsf{in}\ \textsf{inaction})$
   $(\textsf{def}\ D\ \textsf{in}\ P) \mid Q \equiv \textsf{def}\ D\ \textsf{in}\ (P \mid Q) \qquad$ *if* $\mathrm{ft}(\textsf{def}\ D\ \textsf{in}\ \textsf{inaction}) \cap \mathrm{ft}(Q) = \emptyset$
5. $M \equiv M'$ *if M is a permutation of M'*,
   $D \equiv D'$ *if D is a permutation of D'*

Reduction rules drive the computation consuming *redexes* (object/message pairs or instances), replacing them for their *contractions.* Reduction does not necessarily contract a term. For example, a process of the form $X[\tilde{v}]$ expands into the body P of $X(\tilde{x}) = P$ with the parameters $\tilde{x}$ substituted for the values $\tilde{v}$.

A message $a!l_i[\tilde{v}]$ selects the method l_i in an object $a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$; the result is the process P_i where the values of the expressions in $\tilde{v}$ are bound to the variables in $\tilde{x}_i$. This form of reduction is called *communication.*

$$[\textsc{Comm}] \qquad a!l_i[\tilde{v}] \mid a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\} \to \{\tilde{v}/\tilde{x}_i\}P_i$$

Another form of reduction results when we instantiate, $X[\tilde{v}]$, a template definition $X(\tilde{x}) = P$. This results in the process $\{\tilde{v}/\tilde{x}\}P$. This form of reduction is called *instantiation.*

$$[\textsc{Inst}] \qquad \textsf{def}\ X(\tilde{x}) = P\ \textsf{and}\ D\ \textsf{in}\ X[\tilde{v}] \mid Q \to \textsf{def}\ X(\tilde{x}) = P\ \textsf{and}\ D\ \textsf{in}\ \{\tilde{v}/\tilde{x}\}P \mid Q$$

The following rules allow reduction over concurrent composition, introduction of new names and template definitions:

$$[\textsc{Conc}] \qquad \text{If } P \to P' \text{ then } P \mid Q \to P' \mid Q$$
$$[\textsc{New}] \qquad \text{If } P \to P' \text{ then } \textsf{new}\ x\ P \to \textsf{new}\ x\ P'$$
$$[\textsc{Def}] \qquad \text{If } P \to P' \text{ then } \textsf{def}\ D\ \textsf{in}\ P \to \textsf{def}\ D\ \textsf{in}\ P'$$

Finally, the usual closure rule allows the rewriting of terms to produce redexes:

$$[\textsc{Str}] \qquad \text{If } P' \equiv P \text{ and } P \to Q \text{ and } Q \equiv Q' \text{ then } P' \to Q'$$

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

Error processes. A communication error in TyCO is an object/message pair, such that message application is not defined. Two different reasons may cause the error: the message requests a method of the object but with a wrong number of parameters, or the message requests a method that does not exist in the target object. To deal with non-uniform service availability in concurrent objects, a static notion of error is not suitable. We propose a looser notion of process with a communication error, which is dynamic, such that the situations referred above are no longer considered as errors if the request may be accepted by the object at some time in the future (after changing its state). The notion of communication error needs two auxiliary notions:

Context $\mathbf{C}$ is the concurrent composition of messages and a constant '[]' (called the *hole*).

Filling the hole of a context $\mathbf{C}$ with a process P results in the process $\mathbf{C}[P]$.

Redex a pair object/message sharing the same subject.

If message application is defined then the redex may reduce.

Since an object's location is not unique – what allows the implementation of distributed objects by parallel composition – there may be several redexes with the same message. We call *bad redex* a pair of a distributed object and a message where none of possible redexes may reduce.

DEFINITION 3.5 (REDEXES)

1. *A process of the form* $a?M \mid a!l[\tilde{v}]$ *is an* $al\tilde{v}$-redex *(shortly, redex). The object* $a?M$ *and the message* $a!l[\tilde{v}]$ participate *in the redex. The redex is* good *if the contractum* $M \bullet l[\tilde{v}]$ *is defined; it is* bad *otherwise.*
2. *A process of the form* $\Pi_{i \in I}\, a?M_i \mid a!l[\tilde{v}] \mid Q$*, where* I *is a finite indexing set and* $Q \not\equiv a?M \mid Q'$*, is a* distributed *$al\tilde{v}$-redex.*
3. *A process* P has *a (bad) (distributed) redex* R *if* P *has a subterm* R *not under a prefix.*

Errors are processes able to generate bad redexes that persist throughout a computation. An occasional bad redex is not enough to make the process an error if further computation can consume (at least) one of the parties of the bad redex. Thus, messages that may be accepted at some time in the future do not cause errors: an error is a *persistent bad redex.*

We motivate the rigorous definition with some examples.

EXAMPLE 3.6 *Consider the one-place buffer defined in the introduction.*

1. *Process* $Empty[b] \mid b!think$ *is an error, since the object* b *does not have the method requested in any of its states.*
2. *Processes* $R \stackrel{def}{=} Empty[b] \mid b!read[r]$ *and* $S \stackrel{def}{=} Empty[b] \mid b!write[u] \mid b!write[v]$ *are not errors, since the bad redexes can disappear (if the environment provides the right messages).*
3. *Processes* **new** $b\ R$ *and* **new** $b\ S$ *are erroneous, since both processes are locally deadlocked (the scope of the object's name is restricted); hence the bad redexes become persistent (cf. [5]). However,* **new** $b\ R \mid a!l[b]$ *is not an error, since the name* b *can be extruded.*
4. *Process* **new** $a\ (a?\{l = b!m\}) \mid b?\{n\}$ *is a deadlock, but not an error, since the bad* bm*-redex never appears.*
5. *Process* **def** $X(x,y) = x?\{l_1 = y!l_1 \mid x?\{l_2 = y!l_2 \mid X[x,y]\}$ **in** $X[a,a] \mid a!l_1 \mid a!l_2$ *is not an error, although there is always a bad redex, but with different messages (alternates between labels* l_1 *and* l_2*). The bad redex is recurring, but not persistent.*

DEFINITION 3.7 A process P is an *error*, if

$\exists_{\mathrm{C}}\, \mathrm{C}[P] \twoheadrightarrow Q$, and Q has a bad $al\tilde{v}$-redex, for some a occurring in C only has target of a message, and in a message that does not participate in nor generate the bad $al\tilde{v}$-redex, and

$\forall_{\mathrm{C}}\, \mathrm{C}[Q] \twoheadrightarrow R$, and R has a bad $al\tilde{v}$-redex, for some a occurring in C only has target of a message, and in a message that does not participate in nor generate the bad $al\tilde{v}$-redex.

Errors are processes with bad $al\tilde{v}$-redexes that persist throughout reduction. An occasional bad $al\tilde{v}$-redex is not enough to make the process an error. So, we give messages a chance to find their target, and therefore, we say that this calculus have weak fairness in the treatment of messages.

4 The Abstract Machine

In the sequel we present a formal abstract machine specification for TyCO [10]. We start by introducing some preliminary definitions and the syntactic categories used and then proceed with the proper machine definition.

4.1 Preliminary definitions

We first introduce some terminology.

NOTATION 4.1

- *Let $\tilde{s}$ denote the sequence $s_1 \dots s_k$ $(k \geq 0)$ of elements of some syntactic category, and let $\{\tilde{s}\}$ denote the set $\{s_1, \dots, s_n\}$; furthermore, let ϵ be the empty sequence.*
- *Let A, B range over sets, and let $A \mapsto B$ denote the set of* finite maps *(partial functions with finite domain) f, g from A to B. Finite maps are often written explicitly in the form $\{a_1{:}b_1, \dots, a_k{:}b_k\}$ for $k{\geq}0$. In particular, the empty map is $\{\}$. When $\tilde{a} = a_1 \dots a_k$ and $\tilde{b} = b_1 \dots b_k$, we abbreviate the finite map to $\tilde{a}{:}\tilde{b}$.*

DEFINITION 4.2 (MAPS AND QUEUES)

- *The finite map f* modified *by g has domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and values*

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

- *The following grammar inductively defines the set of queues of elements of the syntactic category α.*

$$Queue(\alpha) \quad ::= \quad \bullet \quad | \quad \alpha :: Queue(\alpha)$$

The polymorphic function $\oplus : Queue(\alpha) \times Queue(\alpha) \mapsto Queue(\alpha)$ merges queues.

Now we define the syntactic categories required by the abstract machine.

DEFINITION 4.3 (SYNTACTIC CATEGORIES)

- *A* Thread *T is an element of the set $Thread = Queue(Proc)$. In particular the processes* **new** *x P and* **def** *D* **in** *P are written as, respectively,* **new** *$x :: P$ and* **def** *$D :: P$. The scope of the new variables and definitions extends as far to the right as possible.*
- *An* Environment *B of a thread is a map from variables to values, i.e., $VBind \stackrel{def}{=} Var \mapsto Val$.*
- *A* Template *A or $(\tilde{x})T$ is a map from variables to threads, $Templ \stackrel{def}{=} Var^* \mapsto Thread$.*
- *A* Method *M is a map from labels to templates, $Meth \stackrel{def}{=} Label \mapsto Templ$.*
- *A* Template definition *K is a map from template variables to pairs composed of a set of variable bindings and a template, i.e., $TBind \stackrel{def}{=} TVar \mapsto VBind \times Templ$.*
- *A* Message queue *ms is an element of the set $QMsg \stackrel{def}{=} Queue(Label \times Name^*)$.*
- *An* Object queue *Ms is an element of the set $QMeth \stackrel{def}{=} Queue(VBind \times Meth)$.*
- *A* Channel *Q is an element of the set $QBind \stackrel{def}{=} Name \mapsto QMsg \times QMeth$.*
- *A* Run-Queue *R is an element of the set $RunQueue \stackrel{def}{=} Queue(VBind \times Thread)$.*
  This is where threads and their environments are kept, waiting for execution.
- *A machine* State *S is a tuple in the set $State \stackrel{def}{=} TBind \times VBind \times QBind \times Thread \times RunQueue$.*

4.2 Reduction

The TyCO program, which is the initial thread fed to the abstract machine, must obey to the following restrictions: 1) the initial thread is closed for template and value variables; 2) all bound variables in the initial thread are distinct, and; 3) the initial thread is well typed. These conditions can all be statically verified.

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

$$\emptyset, \emptyset, \emptyset, T, \bullet$$

where T is the initial thread. The machine halts when no rule can be applied, that is when a state of the form below is reached.

$$\_, \_, \_, \bullet, \bullet$$

Rules transform states into states. In the sequel, comments accompanying rules indicate conditions that must be met for the reductions to be successful. These are mostly ensured by the assumptions we make about the initial thread.

The following rules inductively define the abstract machine.

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

$$K, \_, Q, \bullet, (B, T) :: R \longrightarrow K, B, Q, T, R \qquad \text{(Sched)}$$

The Inact rule just skips any **inaction** process in the current thread.

$$K, B, Q, \textbf{inaction} :: T, R \longrightarrow K, B, Q, T, R \qquad \text{(Inact)}$$

The Scop rule introduces a new local variable within a thread. For a thread **new** $x :: T$, 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 T.

$$\frac{a \notin \mathrm{dom}(Q)}{K, B, Q, \textbf{new}\ x :: T, R \longrightarrow K, B + \{x : a\}, Q + \{a : (\bullet, \bullet)\}, T, R} \qquad \text{(Scop)}$$

Comment. By the variable renaming assumption $x \notin \mathrm{dom}(B)$.

The Def rule introduces new template threads bound to template variables. Given a thread **def** $D :: T$, for each definition $X(\tilde{x}) = T'$ in D, the rule adds a bind $X : (B, (\tilde{x})T')$. Execution continues with thread T.

$$K, B, Q, \textbf{def}\ D :: T, R \longrightarrow K + \{X : (B, (\tilde{x})T') \mid X(\tilde{x}) = T' \in D\}, B, Q, T, R \qquad \text{(Def)}$$

Comment. By the variable renaming assumption $\mathrm{dom}(K) \cap \mathrm{dom}(D) = \emptyset$.

The Inst rule instantiates a new thread from a template definition. Given a process $X[\tilde{v}]$ where the template X is defined as $X(\tilde{x}) = T'$, this rule creates a new entry in the run-queue with the thread T' and the variable bindings $B' + \{\tilde{x} : B(\tilde{v})\}$. Notice that B' are the free variable bindings saved at the time the definition of X was found.

$$\frac{K(X) = (B', (\tilde{x})T')}{K, B, Q, X[\tilde{v}] :: T, R \longrightarrow K, B, Q, T, R :: (B' + \{\tilde{x} : B(\tilde{v})\}, T')} \qquad \text{(Inst)}$$

Comment. Since programs are closed for template-variables, $X \in \mathrm{dom}(K)$. Similarly, for value variables, $\{\tilde{v}\} \subseteq \mathrm{dom}(B)$.

Object selection We define a function $\phi_o : \text{Label}\times\text{QMeth}\times\text{QMeth} \mapsto (\text{VBind}\times\text{Meth})\times\text{QMeth}$, such that $\phi_o(l, Ms, \bullet)$ finds the first object closure (B, M) such that $l \in \text{dom}(M)$.

$$
\begin{array}{rcl}
\phi_o(l, \bullet, \_) & = & \bot \\
\phi_o(l, (B,M)::Ms, Ms') & = & \begin{cases} ((B,M), Ms' \oplus Ms) & \text{if } l \in \text{dom}(M) \\ \phi_o(l, Ms, Ms'::(B,M)) & \text{otherwise} \end{cases}
\end{array}
$$

Message selection We define a function $\phi_m : \text{Meth}\times\text{QMsg}\times\text{QMsg} \mapsto (\text{Label}\times\text{Name}^*)\times\text{QMsg}$, such that $\phi_m(M, ms, \bullet)$ finds the first communication $l[\tilde{v}]$ such that $l \in \text{dom}(M)$.

$$
\begin{array}{rcl}
\phi_m(M, \bullet, \_) & = & \bot \\
\phi_m(M, l[\tilde{v}]::ms, ms') & = & \begin{cases} (l[\tilde{v}], ms' \oplus ms) & \text{if } l \in \text{dom}(M) \\ \phi_m(M, ms, ms'::l[\tilde{v}]) & \text{otherwise} \end{cases}
\end{array}
$$

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. In both cases execution continues with the current thread.

$$
\frac{B(x) = a \qquad Q(a) = (ms, Ms) \qquad \phi_o(l, Ms, \bullet) = ((B', M), Ms') \qquad M(l) = (\tilde{x})T'}{K, B, Q, x!l[\tilde{v}]::T, R \longrightarrow K, B, Q + \{a : (ms, Ms')\}, T, R::(B' + \{\tilde{x} : B(\tilde{v})\}, T')} \tag{RedMsg}
$$

$$
\frac{B(x) = a \qquad Q(a) = (ms, Ms) \qquad \phi_m(M, ms, \bullet) = (l[\tilde{v}], ms') \qquad M(l) = (\tilde{x})T'}{K, B, Q, x?M::T, R \longrightarrow K, B, Q + \{a : (ms', Ms)\}, T, R::(B + \{\tilde{x} : \tilde{v}\}, T')} \tag{RedObj}
$$

Comment. Since threads are closed, $x \in \text{dom}(B)$; since programs are well typed, $l \in \text{dom}(M)$, and $\text{length}(\tilde{x}) = \text{length}(\tilde{v})$; since threads are closed and well-typed, $a \in \text{dom}(Q)$; so, the binding must have been placed by the Scop-rule.

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.

$$
\frac{B(x) = a \qquad Q(a) = (ms, Ms) \qquad \phi_o(l, Ms, \bullet) = \bot}{K, B, Q, x!l[\tilde{v}]::T, R \longrightarrow K, B, Q + \{a : (ms::l[B(\tilde{v})], Ms)\}, T, R} \tag{QueueMsg}
$$

$$
\frac{B(x) = a \qquad Q(a) = (ms, Ms) \qquad \phi_m(ms, M, \bullet) = \bot}{K, B, Q, x?M::T, R \longrightarrow K, B, Q + \{a : (ms, Ms::(B, M))\}, T, R} \tag{QueueObj}
$$

Comment. Since threads are closed, $x \in \text{dom}(B)$; also $a \in \text{dom}(Q)$ since threads are closed and well-typed and so the binding must have been placed by the Scop-rule.

The abstract machine just presented grows from Turner's abstract machine for Pict [17], but modifies it in the following major ways:

1. objects are first class entities and substitute input processes. They are more efficient than Pict's [20] both in reduction and heap usage;
2. we use recursion instead of replication for persistence. This allows a cleaner design of the abstract machine – no need for distinct ? and ?* rules – and allows a more rational heap usage;

3. we introduce a new syntactic category – the *Thread* – that represents the basic schedulable and runnable block in the abstract machine. Threads are identified as bodies of template definitions or method implementations;

4. threads cannot be suspended. With this property, our objects are very akin to actors and provide a good model for object oriented concurrent languages [2, 3]. This choice, along with the previous item, also simplifies the treatment of local bindings, introduced with **new** statements, and the management of environments.

5 Comparison with Uniform TyCO

We have seen that from the programming point of view the use of non-uniform types, as opposed to the uniform record types, for representing objects allows a more intuitive and expressive definition of objects and in general object-based data structures. This is a result of the far richer behaviours allowed by the non-uniform type system.

From an implementation point of view also, there are several advantages in using non-uniform types. The discussion that follows steams from our experience in the implementation of TyCO with uniform record types [11].

Comparison with Uniform TyCO. We observe that the abstract machine presented in the previous section is exactly that of uniform TyCO but with distinct definitions for the functions ϕ_o and ϕ_m. In uniform TyCO both functions would be defined simply as:

$$\phi_o(l,\bullet,\_) = \perp \qquad\qquad \phi_m(M,\bullet,\_) = \perp$$
$$\phi_o(l,(B,M)::Ms,\_) = ((B,M),Ms) \qquad \phi_m(M,l[\tilde{v}]::ms,\_) = (l[\tilde{v}],ms)$$

In fact, in the uniform case the abstract machine preserves the invariant that at any moment in a computation the queue associated to a name is either empty or has only messages or has only objects. So, in this case, to try to reduce say a message, we just need to find out if the queue holds objects. Moreover, since record types are uniform we know that any object in the queue will service the message.

In the non-uniform case the invariant no longer holds. We can guarantee however that there are no redexes in the queues. In terms of the implementation we need to keep track of the messages in the queue, or more specifically, their labels. When an object arrives for the queue one must check whether there is a message in the queue that can be serviced by the object.

A similar situation occurs with objects in a queue. Here we must keep track of the label maps for the method collections of the objects in the queue. When a message arrives we must match its label with the maps for the objects to find one that may service the request.

The fact that adapting the abstract machine for uniform TyCO to the non-uniform scheme takes such a small and localised effort gives us some hope that we may profit from the current implementation of TyCO, while keeping it efficient and compact.

Nevertheless, an efficient implementation of channels is crucial and this is a matter of compromise between a faster implementation that uses more heap space per channel but implements queue access in a more efficient way and a slower, space conservative implementation that uses basic sequential search on the queues to find adequate redexes.

Code Size. Programming using non-uniform types can frequently result in much shorter code both in the source programs as well as in the final byte code or other executable format. This is due to the fact that with non-uniform types an object's implementation is typically divided into several instances each representing distinct aspects of the same object, reflecting the dynamic changes in the object's internal state. The objects typically go through a *morphing experience* changing their interfaces dynamically. With uniform TyCO programming such objects requires some extra work and usually involves:

Table 1: Assembler code size of the examples

Example	Uniform	Non-uniform
Buffer	51	30
ATM	273	270
POP3	164	156

1. using distinct channels for each of the object's interfaces since the type system does not allow objects with distinct interfaces at the same channel, and;
2. alternatively, the object may present at all times the same interface, independent of its internal state. It is up to the programmer then to decide what to do when a method is invoked and the object is not in a suitable state (e.g., the Buffer example in section 2). This typically makes the programs cumbersome to read and substantially longer than a corresponding non-uniform implementation.

Table 1 presents the code size for the examples of Section 2.

Heap Usage. Even in cases where the difference in code size is not significant (e.g., the ATM and POP3 examples in section 2) there are two important advantages in non-uniform types:

1. we use a single channel for all of the object's dynamic instances. In the uniform case, a channel must be used for each distinct dynamic instance of an object, despite they are really different aspects of the same object. Thus we can use substantially less channel space in the heap in the non-uniform case;
2. since each dynamic aspect of an object typically features a small subset of the object's total interface and also of the object's free variables, the heap frames that we allocate for each of these objects are smaller than those in the uniform case. Again we can optimise heap usage.

An Example Optimisation linear channels [7], namely those of multiplicity 1 can be highly optimised if using non-uniform types. For example, assume you have the following chain of synchronisations:

$$\textsf{new}\ a\ x![va] \mid a?l = \textsf{new}\ b\ y![ub] \mid b?k = \textsf{new}\ c\ z![wc] \mid c?j = ...$$

Each successive synchronisation is guarded by the previous one. This means that once a, a linear channel, is used for synchronisation it can be discarded and theoretically we could use it again in the next synchronisation instead of allocating a new channel b (and c for that matter):

$$\textsf{new}\ a\ x![va] \mid a?l = y![ua] \mid a?k = z![wa] \mid a?j = ...$$

However, the uniform type system would prevent such usage since the types for the objects do not match (labels l, k and j prevent it). With non-uniform types this optimisation poses no problems whatsoever. We can perform all nested synchronisations with a single channel (here a) that behaves in a *quasi-linear* fashion [8]. This optimisation should greatly improve performance mostly by reducing the amount of heap required for an application. Synchronisation on a linear channel is a pervasive operation in most concurrent programs.

Acknowledgements

This work was partially supported by projects LogComp (PRAXIS XXI contract 2/2.1/TIT/1658/95) DiCoMo (contract PRAXIS/P/EEI/12059/98), and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

References

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

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

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

[4] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[5] G. Boudol. Typing the use of resources in a concurrent calculus. In *Advances in Computing Science*, volume LNCS 1345, pages 239–253. Springer-Verlag, 1997.

[6] N. Kobayashi and A. Yonezawa. Towards foundations of concurrent object-oriented programming - types and language design. *Theory and Practice of Object Systems*, 1(4), 1995.

[7] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. In *ACM Symposium on Principles of Programming Languages*, 1996.

[8] N. Kobayashi. Quasi-Linear Types. In *ACM Symposium on Principles of Programming Languages*, 1999.

[9] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *E-COOP'91*, volume 512 of *LNCS*, pages 133–147. Springer Verlag, 1991.

[10] L. Lopes and V. Vasconcelos. An Abstract Machine for an Object-Calculus. Technical report, DCC-FC & LIACC, Universidade do Porto, May 1997.

[11] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *PPDP'99*, volume 1072 of *LNCS*, pag. 244–260, Springer Verlag, 1999.

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

[13] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, Springer-Verlag, 1993.

[14] Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[15] António Ravara and Vasco T. Vasconcelos. Behavioural Types for a Calculus of Concurrent Objects. In *Euro-Par'97*, volume 1300 of *LNCS*, pages 554–561. Springer Verlag, 1997.

[16] António Ravara and Vasco T. Vasconcelos. Typing Non-uniform Concurrent Objects. Submitted for publication, 1999.

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

[18] 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.

[19] V. Vasconcelos and R. Bastos. Core-TyCO, The Language Definition, Version 0.1. Technical report DI/FCUL TR 98–03, Department of Computer Science, University of Lisbon, 1998.

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