Core-TyCO
The Language Definition Version 0.1

Vasco T. Vasconcelos
Rui Bastos

DI–FCUL TR–98–3

March 1998

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

This is the second report on TyCO [3], a (still) experimental strongly and implicitly typed concurrent object oriented programming language based on a predicative polymorphic calculus of objects [4, 5], featuring asynchronous messages, objects, and process declarations, together with a predicative polymorphic typing assignment system assigning monomorphic types to variables and polymorphic types to process variables.

Sections 1 and 2 define the syntax and static semantics of the language, both in the style of Standard ML [2]. Dynamic semantics is the subject of Section 3, presented along the lines of the π-calculus [1].

Contents

1 Syntax

1.1 Reserved words

Figure 1 lists the *reserved words* of core-TyCO. They may not be used as identifiers.

```
and  branch  def  else  if  in  inaction
      into  let  new  not  or  then
        !  ?  |  {  }  [  ]  (  )  ,  _
   +  -  *  /  %  =  <>  >  >=  <  <=  ^
```

Figure 1: Reserved Words

c or *const*	$\in$	Const	constants (Integer, Boolean, and String)
a, x or *var*	$\in$	Var	variables
v or *val*	$\in$	Const $\cup$ Var	values
l or *label*	$\in$	Label	labels
X or *procvar*	$\in$	ProcVar	process variables

Figure 2: Classes of Identifiers

1.2 Identifiers

The *classes of identifiers* for core-TyCO are shown in Figure 2.

An *integer constant* (decimal notation only) is an optional negation symbol (`-`) followed by a non-empty sequence of decimal digits `0`–`9`. A *boolean constant* is either `true` or `false`. A *string constant* is a sequence, between quotes (`"`), of zero or more characters.

1.3 Grammar

The *phrase classes* for core-TyCO are shown in Figure 3 and the *grammatical rules* in Figure 4. Brackets $\langle\ \rangle$ enclose optional phrases.

program	$\in$	Program	programs
P, Q, R or *proc*	$\in$	Proc	processes
D or *dec*	$\in$	Dec	declarations
bind	$\in$	Bind	process bindings
multbind	$\in$	MultBind	sequence of bindings
M or *method*	$\in$	Method	methods
M^+ or *methrow*	$\in$	MethRow	method rows
e or *exp*	$\in$	Exp	expressions
expseq	$\in$	Exp^+	sequence of expressions
varseq	$\in$	Var^+	sequence of variables
valseq	$\in$	Val^+	sequence of values
binop	$\in$	BinOp	binary operators
unop	$\in$	UnOp	unary operators

Figure 3: Phrase Classes

The scope of `new` extends as far to the right as possible, and the operator `|` takes precedence over `def`–`in`, so, for example, `def` D `in new` x P `|` Q means `def` D `in (new` x `(`P `|` Q`))`. The precedence and associativity of the remaining operators (including arithmetical and logical) is standard.

program	::=	*proc*	program
proc	::=	*var* `!` *label* `[` ⟨*expseq*⟩ `]`	message
		var `?` `{` ⟨*methrow*⟩ `}`	object
		`new` *var* *proc*	scope restriction
		`def` *dec* `in` *proc*	local declaration
		procvar `[` ⟨*expseq*⟩ `]`	process instantiation
		proc `|` *proc*	parallel composition
		`if` *exp* `then` *proc* `else` *proc*	conditional
		`inaction`	inaction
		`(` *proc* `)`	
dec	::=	*multbind*	sequence of bindings
multbind	::=	*bind* ⟨`and` *multbind*⟩	multiple binding
bind	::=	*procvar* `(` ⟨*varseq*⟩ `)` `=` *proc*	process binding
methrow	::=	*method* ⟨`,` *methrow*⟩	method row
method	::=	*label* `(` ⟨*varseq*⟩ `)` `=` *proc*	method
exp	::=	*exp* *binop* *exp*	infixed expression
		unop *exp*	prefixed expression
		val	value
		`(` *exp* `)`	
expseq	::=	*exp* ⟨`,` *expseq*⟩	sequence of expressions
varseq	::=	*var* ⟨`,` *varseq*⟩	sequence of variables
binop	::=	`+` | `-` | `*` | `/` | `%` | `^` |	
		`=` | `<>` | `>` | `>=` | `<` | `<=` |	
		`and` | `or`	
unop	::=	`-` | `not`	

Figure 4: Grammar

1.4 Syntactic restrictions

1. No label may appear twice in the same method row.
2. No method or process binding parameter list may contain the same variable twice.
3. No sequence of bindings may contain the same process variable twice.

1.5 Comments

A *comment* is any character sequence beginning with `--` and extending to the end of the same line.

1.6 Derived forms for processes

A list of *derived forms* built from the primitives of the core language is shown in Figure 5, where variable z is fresh. Symbol $\Longrightarrow$ means 'rewrites to'.

2 Static semantics

2.1 Simple objects

The *simple objects* for the static semantics are defined in Figure 6.

2.2 Compound objects

When A and B are sets, $A \mapsto B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain of a finite map f is denoted $\mathrm{dom}(f)$.

Figure 7 shows the compound objects for the static semantics.

2.3 Operations on finite maps

A finite map will often be written explicitly in the form $\{a_1 : b_1, \ldots, a_n : b_n\}$, for $n \geq 0$; in particular, the empty map is $\{\}$. When $a^n = a_1 \cdots a_n$ and $b^n = b_1 \cdots b_n$, we abbreviate the above finite map to $\{a^n : b^n\}$.

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

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

When f is a finite map and A a set, $f \setminus A$, called f *restricted by* A, is the finite map with domain $\mathrm{dom}(f) \setminus A$ and values

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

2.4 Recursive types, infinite trees, and typing compatibility

Types are interpreted as regular infinite trees. A translation $()^*$ from types to infinite trees is defined as follows.

$$
\begin{array}{rcl}
t^* & = & t \\
\varrho^* & = & \{l : \alpha_1^* \cdots \alpha_n^* \mid l : \alpha_1 \cdots \alpha_n \in \varrho\} \\
\mu t.\alpha^* & = & \mathbf{fix}(\lambda\rho.\alpha^*[\rho/t])
\end{array}
$$

Two types α and β are *equivalent*, denoted by $\alpha \approx \beta$, iff $\alpha^* = \beta^*$. Two typings Γ and Δ are *compatible*, denoted by $\Gamma \asymp \Delta$, iff whenever $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$ then $\Gamma(a) \approx \Delta(a)$. Similarly, two record types ϱ and ϱ' are compatible, $\varrho \asymp \varrho'$, iff whenever $l \in \mathrm{dom}(\varrho) \cap \mathrm{dom}(\varrho')$ then $\varrho(l) = \alpha_1 \cdots \alpha_n$, $\varrho'(l) = \beta_1 \cdots \beta_n$, and $\alpha_i \approx \beta_i$, for $1 \leq i \leq n$.

2.5 Free type variables

The set of *free type variables* in the various semantic objects is inductively defined as follows.

$$
\begin{array}{rcll}
\mathrm{ftv}(t) & = & \{t\} & \\
\mathrm{ftv}(\alpha_1 \cdots \alpha_n) & = & \bigcup_{1 \leq i \leq n} \mathrm{ftv}(\alpha_i) & \\
\mathrm{ftv}(\mu t.\alpha) & = & \mathrm{ftv}(\alpha) \setminus \{t\} & \\
\mathrm{ftv}(\forall t^n.\tau) & = & \mathrm{ftv}(\tau) \setminus \{t^n\} & \\
\mathrm{ftv}(f) & = & \bigcup_{a \in \mathrm{dom}(f)} \mathrm{ftv}(a) & \quad \text{(for } f \text{ a finite map)}
\end{array}
$$

2.6 Type schemes, closure, and instances

Two type schemes are considered equal if they can be obtained from each other by renaming and reordering of bound type variables, and deleting type variables from the prefix which do not occur in the body.

A type scheme $\sigma = \forall t^n.\tau$ *generalizes* another type scheme $\sigma' = \forall u^n.\tau'$, written $\sigma \succ \sigma'$, if $\tau' = \tau[\alpha^n/t^n]$, for some α^n, and u^n contains no free type variables of σ.

Let τ be a sequence of types and E an environment. The *closure* of τ with respect to E, $\mathrm{clos}_E(\tau)$, is the type scheme $\forall t^n.\tau$, where $t^n = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(E)$.

When B is a basis whose range contains only type sequences (rather than arbitrary type schemes), $\mathrm{clos}_E(B)$, the closure of B with respect to E, is the basis $\{X : \mathrm{clos}_E(\tau) \mid X : \tau \in B\}$.

2.7 Environment modification and projection

When E is an environment, Γ a typing and B a basis, $E + B$ means $E + (\{\}, B)$, and $E + \Gamma$ means $E + (\Gamma, \{\})$.

Given an environment E, the expression C of E accesses the C component of E, namely, Γ of E means "the typing component of E" and B of E means "the basis component of E".

2.8 Derived forms for types

Notation $\alpha^n \to \beta^m$, for $n, m \geq 0$, is used to emphasize the "functional" nature of some methods, and abbreviates type $\{\mathtt{val} : \alpha^n\{\mathtt{val} : \beta^m\}\}$. For empty type sequences (when n or m are 0) we write () instead of ε.

2.9 Types for primitive operations and objects

Core-TyCO has as predefined the primitive types `int`, `bool`, and `string`, along with the following operations.

$$
\begin{array}{lcllcl}
\mathtt{+} & : & \mathtt{int\ int} & \to & \mathtt{int} \\
\mathtt{-} & : & \mathtt{int\ int} & \to & \mathtt{int} \\
\mathtt{*} & : & \mathtt{int\ int} & \to & \mathtt{int} \\
\mathtt{/} & : & \mathtt{int\ int} & \to & \mathtt{int} \\
\mathtt{\%} & : & \mathtt{int\ int} & \to & \mathtt{int} \\
\mathtt{-} & : & \mathtt{int} & \to & \mathtt{int} \\
\mathtt{\hat{}} & : & \mathtt{string\ string} & \to & \mathtt{string} \\
\mathtt{and} & : & \mathtt{bool\ bool} & \to & \mathtt{bool} \\
\mathtt{or} & : & \mathtt{bool\ bool} & \to & \mathtt{bool} \\
\mathtt{not} & : & \mathtt{bool} & \to & \mathtt{bool}
\end{array}
$$

The usual relational operations are also provided. Currently they may take as arguments only integers.

$$
\begin{array}{lcllcl}
\mathtt{=} & : & \mathtt{int\ int} & \to & \mathtt{bool} \\
\mathtt{<>} & : & \mathtt{int\ int} & \to & \mathtt{bool} \\
\mathtt{<} & : & \mathtt{int\ int} & \to & \mathtt{bool} \\
\mathtt{<=} & : & \mathtt{int\ int} & \to & \mathtt{bool} \\
\mathtt{>} & : & \mathtt{int\ int} & \to & \mathtt{bool} \\
\mathtt{>=} & : & \mathtt{int\ int} & \to & \mathtt{bool}
\end{array}
$$

A basic stream based I/O facility is available by means of an object `io`.

$$
\begin{array}{lcl}
\mathtt{io} & : & \{\mathtt{getb} : () \to \mathtt{bool}, \\
& & \ \mathtt{putb} : \mathtt{bool}, \\
& & \ \mathtt{geti} : () \to \mathtt{int}, \\
& & \ \mathtt{puti} : \mathtt{int}, \\
& & \ \mathtt{gets} : () \to \mathtt{string}, \\
& & \ \mathtt{puts} : \mathtt{string}\}
\end{array}
$$

2.10 Inference rules

Variables $\boxed{\Gamma \vdash \mathit{var} \Rightarrow \mathit{type}}$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x \Rightarrow \Gamma(x)} \tag{1}$$

$$\frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash x \Rightarrow \alpha} \tag{2}$$

Comment:

(2) Allows for free variables in processes.

Sequences of variables $\boxed{\Gamma \vdash \mathit{varseq} \Rightarrow \mathit{typeseq}}$

$$\frac{\Gamma \vdash x \Rightarrow \alpha \qquad \langle \Gamma + \{x : \alpha\} \vdash \mathit{varseq} \Rightarrow \alpha^n \rangle}{\Gamma \vdash x \ \langle \texttt{,}\ \mathit{varseq} \rangle \Rightarrow \alpha \langle \alpha^n \rangle} \tag{3}$$

Comment:

When the option is present, *varseq* is a sequence of n variables.

Expressions $\boxed{\Gamma \vdash \mathit{exp} \Rightarrow \mathit{type}, \Gamma}$

Function typeof, when applied to a constant or a primitive operator, returns its type.

$$\Gamma \vdash c \Rightarrow \mathrm{typeof}(c), \{\} \tag{4}$$

$$\frac{\Gamma \vdash x \Rightarrow \alpha}{\Gamma \vdash x \Rightarrow \alpha, \{x : \alpha\}} \tag{5}$$

$$\frac{\begin{array}{c}\mathrm{typeof}(\mathit{binop}) = \rho_1\ \rho_2 \to \rho \\ \Gamma \vdash \mathit{exp}_1 \Rightarrow \rho_1, \Gamma_1 \qquad \Gamma \vdash \mathit{exp}_2 \Rightarrow \rho_2, \Gamma_2 \qquad \Gamma_1 \asymp \Gamma_2\end{array}}{\Gamma \vdash \mathit{exp}_1\ \mathit{binop}\ \mathit{exp}_2 \Rightarrow \rho, \Gamma_1 + \Gamma_2} \tag{6}$$

$$\frac{\mathrm{typeof}(\mathit{unop}) = \rho \to \rho' \qquad \Gamma \vdash \mathit{exp} \Rightarrow \rho, \Gamma'}{\Gamma \vdash \mathit{unop}\ \mathit{exp} \Rightarrow \rho', \Gamma'} \tag{7}$$

Comments:

(6) By the current definition of binary operators, both parameters always have the same type, so $\rho_1 = \rho_2$.

(7) Both the parameter and the returned result of all current unary operators have the same type, so $\rho = \rho'$.

Sequences of expressions $\boxed{\Gamma \vdash \mathit{expseq} \Rightarrow \mathit{typeseq}, \Gamma}$

$$\frac{\Gamma \vdash \mathit{exp} \Rightarrow \alpha, \Gamma' \qquad \langle \Gamma + \Gamma' \vdash \mathit{expseq} \Rightarrow \alpha^n, \Gamma'' \rangle}{\Gamma \vdash \mathit{exp}\ \langle \texttt{,}\ \mathit{expseq} \rangle \Rightarrow \alpha \langle \alpha^n \rangle, \Gamma' \langle + \Gamma'' \rangle} \tag{8}$$

Comment:

When the option is present, *expseq* is a sequence of n expressions.

Methods $\boxed{E \vdash \mathit{method} \Rightarrow \varrho, \Gamma}$

$$\frac{E\ \langle + \{x^n : \alpha^n\} \rangle \vdash \mathit{proc} \Rightarrow \Gamma}{E \vdash l\texttt{(}\langle x_1 \texttt{,} \ldots \texttt{,} x_n \rangle\texttt{)} \texttt{ = } \mathit{proc} \Rightarrow \{l : \varepsilon \langle \alpha^n \rangle\}, \Gamma \langle \setminus \{x^n\} \rangle} \tag{9}$$

Comment:
When the option is present, the variables x^n don't appear in the resulting typing.

Method rows $\boxed{E \vdash \mathit{methrow} \Rightarrow \varrho, \Gamma}$

$$\frac{E \vdash \mathit{method} \Rightarrow \varrho, \Gamma \qquad E \vdash \mathit{methrow} \Rightarrow \varrho', \Gamma' \qquad \Gamma \asymp \Gamma'}{E \vdash \mathit{method}\ \langle \texttt{,}\ \mathit{methrow}\rangle \Rightarrow \varrho\langle + \varrho'\rangle, \Gamma\langle + \Gamma'\rangle} \tag{10}$$

Comment:
When the option is present, we have $\mathrm{dom}(\varrho) \cap \mathrm{dom}(\varrho') = \emptyset$, by the syntactic restrictions.

Processes $\boxed{E \vdash \mathit{proc} \Rightarrow \Gamma}$

$$E \vdash \texttt{inaction} \Rightarrow \{\} \tag{11}$$

$$\frac{\Gamma \text{ of } E \vdash a \Rightarrow \varrho \qquad \langle \Gamma \text{ of } E \vdash \mathit{expseq} \Rightarrow \alpha^n, \Gamma\rangle \qquad l \in \mathrm{dom}(\varrho) \wedge \varrho \asymp \{l : \varepsilon\langle\alpha^n\rangle\}}{E \vdash a\texttt{!}l\texttt{[}\langle\mathit{expseq}\rangle\texttt{]} \Rightarrow \{a : \varrho\}\ \langle + \Gamma\rangle} \tag{12}$$

$$\frac{\Gamma \text{ of } E \vdash a \Rightarrow \{\}\langle + \varrho\rangle \qquad \langle E \vdash \mathit{methrow} \Rightarrow \varrho, \Gamma \qquad \{a : \varrho\} \asymp \Gamma\rangle}{E \vdash a\texttt{?\{}\langle\mathit{methrow}\rangle\texttt{\}} \Rightarrow \{a : \{\}\langle + \varrho\rangle\}\ \langle + \Gamma\rangle} \tag{13}$$

$$\frac{E \vdash \mathit{proc}_1 \Rightarrow \Gamma_1 \qquad E \vdash \mathit{proc}_2 \Rightarrow \Gamma_2 \qquad \Gamma_1 \asymp \Gamma_2}{E \vdash \mathit{proc}_1 \texttt{ | } \mathit{proc}_2 \Rightarrow \Gamma_1 + \Gamma_2} \tag{14}$$

$$\frac{E + \{x : \alpha\} \vdash \mathit{proc} \Rightarrow \Gamma}{E \vdash \texttt{new}\ x\ \mathit{proc} \Rightarrow \Gamma \setminus \{x\}} \tag{15}$$

$$\frac{B \text{ of } E(X) \succ \varepsilon\langle\alpha^n\rangle \qquad \langle \Gamma \text{ of } E \vdash \mathit{expseq} \Rightarrow \alpha^n, \Gamma\rangle}{E \vdash X\texttt{[}\langle\mathit{expseq}\rangle\texttt{]} \Rightarrow \{\}\langle + \Gamma\rangle} \tag{16}$$

$$\frac{E \vdash \mathit{dec} \Rightarrow B \qquad E + B \vdash \mathit{proc} \Rightarrow \Gamma}{E \vdash \texttt{def}\ \mathit{dec}\ \texttt{in}\ \mathit{proc} \Rightarrow \Gamma} \tag{17}$$

$$\frac{E \vdash \mathit{proc} \Rightarrow \Gamma}{E \vdash \texttt{(}\mathit{proc}\texttt{)} \Rightarrow \Gamma} \tag{18}$$

$$\frac{\begin{array}{c}\Gamma \text{ of } E \vdash \mathit{exp} \Rightarrow \texttt{bool}, \Gamma \\ E \vdash \mathit{proc}_1 \Rightarrow \Gamma_1 \qquad E \vdash \mathit{proc}_2 \Rightarrow \Gamma_2 \qquad \Gamma \asymp \Gamma_1 \asymp \Gamma_2\end{array}}{E \vdash \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{proc}_1\ \texttt{else}\ \mathit{proc}_2 \Rightarrow \Gamma + \Gamma_1 + \Gamma_2} \tag{19}$$

Comments:

(12) Object a must contain (at least) a method l, accepting as arguments n values of types α^n, for $n \geq 0$.

(15) Bound variable x does not appear in the resulting typing.

(16) When the option is present, the instantiation of type schemes allows different occurrences of a process variable to have different types.

Bindings $\boxed{E \vdash \mathit{bind} \Rightarrow B}$

$$\frac{E\ \langle + \{x^n : \alpha^n\}\rangle \vdash \mathit{proc} \Rightarrow \{\}\langle + \{x^n : \alpha^n\}\rangle}{E \vdash X\,\texttt{(}\langle x_1\,\texttt{,}\,\ldots\,\texttt{,}x_n\rangle\texttt{)}\ \texttt{=}\ \mathit{proc} \Rightarrow \{X : \varepsilon\langle\alpha^n\rangle\}} \tag{20}$$

Comment:

In the resulting basis, process variable X always has a type instead of a type scheme.

Multiple bindings $\boxed{E \vdash \mathit{multbind} \Rightarrow B}$

$$\frac{\begin{array}{c} B = B' + B'' \\ E + B \vdash \mathit{bind} \Rightarrow B' \qquad E + B \vdash \mathit{multbind} \Rightarrow B'' \end{array}}{E \vdash \mathit{bind}\ \langle\texttt{and}\ \mathit{multbind}\rangle \Rightarrow B} \tag{21}$$

Comment:

When the option is present, the syntactic restrictions assure that $\mathrm{dom}(B') \cap \mathrm{dom}(B'') = \emptyset$.

Declarations $\boxed{E \vdash \mathit{dec} \Rightarrow B}$

$$\frac{E \vdash \mathit{multbind} \Rightarrow B}{E \vdash \mathit{dec} \Rightarrow \mathrm{clos}_E(B)} \tag{22}$$

Comment:

By rules (20) and (21) we have that basis B contains only types. The closure of B is what allows process variables to be used polymorphically, via rule (16) (in rule (17), *proc* is typed in an environment that already includes the polymorphic basis).

Programs $\boxed{E \vdash \mathit{program} \Rightarrow \Gamma}$

$$E \vdash \mathit{program} \Rightarrow \{\} \tag{23}$$

Comment:

A program must have no free variables.

3 Dynamic semantics

3.1 Free identifiers

A variable x occurs *free* in a process P if x is not in the scope P of a method $l\texttt{(}\ldots\,x\,\ldots\texttt{)}\ \texttt{=}\ P$, a process binding $X\texttt{(}\ldots\,x\,\ldots\texttt{)}\ \texttt{=}\ P$, or a scope restriction

$\texttt{new}\ x\ P$; otherwise it occurs *bound*. The set of free variables in a process P is denoted by $\mathrm{fn}(P)$.

A process variable X occurs free in a process if X is not in the scope $P, Q\langle, D\rangle$ of a declaration $\texttt{def}\ X\texttt{(...)} = Q\ \langle\texttt{and}\ D\rangle\ \texttt{in}\ P$; otherwise it occurs bound. The set of free process variables in a process P is denoted by $\mathrm{fv}(P)$.

3.2 Variable substitution and α-conversion

The simultaneous *substitution of free variables* x^n in a process P by values v^n, for $n \geq 0$, and provided that each x does not appear twice in x^n, is denoted by $P[v^n/x^n]$.

A process P is *α-convertible* to Q if Q results from P by a series of changes of bound variables and bound process variables.

3.3 Structural congruence

Processes $\boxed{\mathit{proc} \equiv \mathit{proc}}$

$$\frac{P \text{ is } \alpha\text{-convertible to } Q}{P \equiv Q} \tag{1}$$

$$P \mid Q \equiv Q \mid P \tag{2}$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \tag{3}$$

$$P \mid \texttt{inaction} \equiv P \tag{4}$$

$$\texttt{new}\ x\ \texttt{inaction} \equiv \texttt{inaction} \tag{5}$$

$$\texttt{new}\ x\ \texttt{new}\ y\ P \equiv \texttt{new}\ y\ \texttt{new}\ x\ P \tag{6}$$

$$\frac{x \notin \mathrm{fn}(Q)}{(\texttt{new}\ x\ P) \mid Q \equiv \texttt{new}\ x\ P \mid Q} \tag{7}$$

$$\texttt{def}\ D\ \texttt{in}\ \texttt{inaction} \equiv \texttt{inaction} \tag{8}$$

$$\frac{x \notin \mathrm{fn}(D)}{\texttt{def}\ D\ \texttt{in}\ \texttt{new}\ x\ P \equiv \texttt{new}\ x\ \texttt{def}\ D\ \texttt{in}\ P} \tag{9}$$

$$\frac{\mathrm{fv}(D) \cap \mathrm{fv}(Q) = \emptyset}{(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \equiv \texttt{def}\ D\ \texttt{in}\ (P \mid Q)} \tag{10}$$

Method rows $\boxed{methrow \equiv methrow}$

$$\frac{M'^{+} \text{ is a permutation of } M^{+}}{M'^{+} \equiv M^{+}} \tag{11}$$

Declarations $\boxed{dec \equiv dec}$

$$\frac{D' \text{ is a permutation of } D}{D' \equiv D} \tag{12}$$

3.4 Reduction rules

Expressions $\boxed{exp \rightarrow val}$

$$val \rightarrow val \tag{1}$$

$$\frac{exp_1 \rightarrow c_1 \qquad exp_2 \rightarrow c_2}{exp_1\ binop\ exp_2 \rightarrow c_1\ binop\ c_2} \tag{2}$$

$$\frac{exp \rightarrow c}{unop\ exp \rightarrow unop\ c} \tag{3}$$

Comment:
(1) Constants and variables reduce to themselves.

Sequences of expressions $\boxed{expseq \rightarrow valseq}$

$$\frac{exp \rightarrow v \qquad \langle expseq \rightarrow v^n \rangle}{exp\ \langle \texttt{,}\ expseq \rangle \rightarrow v \langle v^n \rangle} \tag{4}$$

Processes $\boxed{proc \rightarrow proc}$

$$\frac{\langle expseq \rightarrow v^n \rangle}{a\texttt{!}l\texttt{[}\langle expseq \rangle\texttt{]}\ \texttt{|}\ a\texttt{?\{}l\texttt{(}\langle x_1\texttt{,} \ldots \texttt{,}x_n \rangle\texttt{)}\ \texttt{=}\ P\ \langle \texttt{,}\ methrow \rangle \texttt{\}} \rightarrow P\langle [v^n/x^n] \rangle} \tag{5}$$

$$\frac{\langle expseq \rightarrow v^n \rangle}{\begin{array}{r}\texttt{def}\ X\texttt{(}\langle x_1\texttt{,} \ldots \texttt{,}x_n \rangle\texttt{)}\ \texttt{=}\ P\ \langle \texttt{and}\ D \rangle\ \texttt{in}\ X\texttt{[}\langle expseq \rangle\texttt{]}\ \langle \texttt{|}\ Q \rangle \rightarrow \\ \texttt{def}\ X\texttt{(}\langle x_1\texttt{,} \ldots \texttt{,}x_n \rangle\texttt{)}\ \texttt{=}\ P\ \langle \texttt{and}\ D \rangle\ \texttt{in}\ P\langle [v^n/x^n] \rangle\ \langle \texttt{|}\ Q \rangle\end{array}} \tag{6}$$

$$\frac{P \rightarrow P'}{P\ \texttt{|}\ Q \rightarrow P'\ \texttt{|}\ Q} \tag{7}$$

$$\frac{P \to P'}{\texttt{new}\ x\ P \to \texttt{new}\ x\ P'} \tag{8}$$

$$\frac{P \to P'}{\texttt{def}\ D\ \texttt{in}\ P \to \texttt{def}\ D\ \texttt{in}\ P'} \tag{9}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \tag{10}$$

References

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

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

[3] Vasco T. Vasconcelos. TyCO: the language definition, version 0.0. Keio University, July 1993.

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

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

`new` x_1`,...,`x_n *proc*	$\Longrightarrow$	`new` x_1 $\cdots$ `new` x_n *proc*
a`?(`$\langle$*varseq*$\rangle$`) =` *proc*	$\Longrightarrow$	a`?{val(`$\langle$*varseq*$\rangle$`) =` *proc*`}`
a`![`$\langle$*expseq*$\rangle$`]`	$\Longrightarrow$	a`!val[`$\langle$*expseq*$\rangle$`]`
l`(... _ ...) =` *proc*	$\Longrightarrow$	l`(...` z `...) =` *proc*
X`(... _ ...) =` *proc*	$\Longrightarrow$	X`(...` z `...) =` *proc*
`if` *exp* `then` *proc*	$\Longrightarrow$	`if` *exp* `then` *proc* `else inaction`
`branch` a`!`$\langle l\rangle$`[`$\langle$*expseq*$\rangle$`]` `into {`$\langle$*methrow*$\rangle$`}`	$\Longrightarrow$	`new` z a`!`$\langle l\rangle$`[`$\langle$*expseq*`,`$\rangle z$`] |` z`?{`$\langle$*methrow*$\rangle$`}`
`branch` X`[`$\langle$*expseq*$\rangle$`]` `into {`$\langle$*methrow*$\rangle$`}`	$\Longrightarrow$	`new` z X`[`$\langle$*expseq*`,`$\rangle z$`] |` z`?{`$\langle$*methrow*$\rangle$`}`
`let` *varseq* `=` a`!`$\langle l\rangle$`[`$\langle$*expseq*$\rangle$`]` `in` *proc*	$\Longrightarrow$	`branch` a`!`$\langle l\rangle$`[`$\langle$*expseq*$\rangle$`]` `into {val(`*varseq*`) =` *proc*`}`
`let` *varseq* `=` X`[`$\langle$*expseq*$\rangle$`]` `in` *proc*	$\Longrightarrow$	`branch` X`[`$\langle$*expseq*$\rangle$`]` `into {val(`*varseq*`) =` *proc*`}`

Figure 5: Derived Forms

$$
\begin{array}{rcl}
u, t \text{ or } \mathit{typevar} & \in & \text{TypeVar} \\
\mathit{typevarseq} & \in & \text{TypeVarSeq} = \text{TypeVar}^n \\
\rho \text{ or } \mathit{primtype} & \in & \text{PrimType} = \{\texttt{int}, \texttt{bool}, \texttt{string}\}
\end{array}
$$

Figure 6: Simple Semantic Objects

$$
\begin{array}{rcl}
\alpha, \beta & \in & \text{Type} = \text{PrimType} \cup \text{TypeVar} \cup \text{RcdType} \cup \text{RecType} \\
\tau \text{ or } \alpha^n & \in & \text{TypeSeq} = \text{Type}^n \\
\varepsilon & \in & \text{Type}^0 \\
\varrho & \in & \text{RcdType} = \text{Label} \mapsto \text{TypeSeq} \\
\mu t.\alpha & \in & \text{RecType} = \text{TypeVar} \times \text{Type} \\
\sigma \text{ or } \forall t^n.\tau & \in & \text{TypeScheme} = \bigcup_{n \geq 0} \text{TypeVar}^n \times \text{TypeSeq} \\
\Gamma, \Delta & \in & \text{Typing} = \text{Var} \mapsto \text{Type} \\
B & \in & \text{Basis} = \text{ProcVar} \mapsto \text{TypeScheme} \\
E & \in & \text{Env} = \text{Typing} \times \text{Basis}
\end{array}
$$

Figure 7: Compound Semantic Objects