Virtual Machines for Process Calculi

Vasco T. Vasconcellos
University of Lisbon

Joint work with

Luís Lopes and Fernando Silva Silva
University of Porto
Goals

- Technology to run process-calculi based programming languages on multiprocessors;
- Allow for multiple idioms: pi, TyCO, asynchronous, synchronous, replication, recursion, join, blue;
- Design a series of increasingly specific machines, each one sound wrt to the previous;
- Prepare ground for distribution and code mobility.
Outline

• Four machines:
  – Threaded TyCO;
  – Simple Machine;
  – Environment Machine;
  – Byte Code Machine;

• Distribution and Mobility

• TyCOProgramming language, tycoc, tyco
Threaded TyCO—Syntax

\[ T ::= [\vec{I}] \quad \text{Thread} \]

\[ I ::= S \mid X\langle \vec{x} \rangle \mid \text{new } x \quad \text{Instruction} \]

\[ S ::= x = M \mid x.l\langle \vec{x} \rangle \mid X = A \quad \text{Store} \]

\[ M ::= \{ l = A \} \quad \text{Method map} \]

\[ A ::= (\vec{x})T \quad \text{Abstraction} \]
Threaded TyCO

Machine states.

\[
\text{run } \tilde{T} \text{ in } \tilde{S}
\]

Structural congruence

\[
\tilde{\alpha} \equiv \tilde{\alpha}' \quad \text{if } \tilde{\alpha} \text{ is a permutation of } \tilde{\alpha}'
\]

\[
\; , \tilde{T} \equiv \tilde{T}
\]
Threaded TyCO—Reduction

\[
\text{run } [x.l\langle\bar{y}\rangle; \bar{I}], \bar{T} \text{ in } x = M, \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T}, M.l\langle\bar{y}\rangle \text{ in } \tilde{S}
\]

\[
\text{run } [x = M; \bar{I}], \bar{T} \text{ in } x.l\langle\bar{y}\rangle, \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T}, M.l\langle\bar{y}\rangle \text{ in } \tilde{S}
\]

\[
\text{run } [X\langle\bar{y}\rangle; \bar{I}], \bar{T} \text{ in } X = A, \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T}, A\langle\bar{y}\rangle \text{ in } X = A, \tilde{S}
\]

\[
\text{run } [x = M; \bar{I}], \bar{T} \text{ in } \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T} \text{ in } x = M, \tilde{S}
\]

\[
\text{run } [x.l\langle\bar{y}\rangle, \bar{I}], \bar{T} \text{ in } \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T} \text{ in } x.l\langle\bar{y}\rangle, \tilde{S}
\]

\[
\text{run } [X = A; \bar{I}], \bar{T} \text{ in } \tilde{S} \rightarrow \text{run } [\bar{I}], \tilde{T} \text{ in } X = A, \tilde{S}
\]

\[
\text{run } \text{new } x; \bar{I}, \tilde{T} \text{ in } \tilde{S} \rightarrow \text{run } \{y/x\}\bar{I}, \tilde{T} \text{ in } \tilde{S} \quad \text{if } y \text{ not free in }
\]
Theorem 1 (Soundness) 1. If \( C \to C' \), then either
\[
\llbracket C \rrbracket \equiv \llbracket C' \rrbracket \text{ or } \llbracket C \rrbracket \to \llbracket C' \rrbracket;
\]
2. If \textbf{run} \( T_0 \textbf{ in } \varepsilon \downarrow C \) then \( \llbracket C \rrbracket \not\vdash \).
Environment Machine

An environment \( e \) is a map from object tags into object tags. The store becomes a map \( s \) from thread tags \( X \) to abstraction closures \( A e \), and from object tags \( x \) to queues \( q \) of method closures \( M e \) or of messages contents \( l \langle x \rangle \).
Environment Machine—Reduction I

\[ \text{run } [x.l\langle \vec{y} \rangle; \vec{I}]e, \tilde{c} \text{ in } s \rightarrow \text{run } [\vec{I}]e, \tilde{c}, Te' \{ \vec{z} := e(\vec{y}) \} \text{ in } s \{ e(x) := q \} \]

\begin{align*}
\text{if } s(e(x)) &= Me' : q, M.l = (\vec{z})T \\
\text{run } [x = M; \vec{I}]e, \tilde{c} \text{ in } s &\rightarrow \text{run } [\vec{I}]e, \tilde{c}, Te \{ \vec{z} := \vec{y} \} \text{ in } s \{ e(x) := q \} \\
\text{if } s(e(x)) &= l\langle \vec{y} \rangle : q, M.l = (\vec{z})T \\
\text{run } [X\langle \vec{y} \rangle; \vec{I}]e, \tilde{c} \text{ in } s &\rightarrow \text{run } [\vec{I}]e, \tilde{c}, Te' \{ \vec{z} := e(\vec{y}) \} \text{ in } s \\
\text{if } s(X) &= ((\vec{z})T)e' \\
\end{align*}
Environment Machine—Reduction II

\[
\text{run } [x = M; \overrightarrow{I}e, \tilde{c} \text{ in } s \rightarrow \text{run } [\overrightarrow{I}]e, \tilde{c} \text{ in } s\{e(x) := q : Me}\]
\text{ if } s(e(x)) = q
\]

\[
\text{run } [x \cdot \langle y \rangle, \overrightarrow{I}e, \tilde{c} \text{ in } s \rightarrow \text{run } [\overrightarrow{I}]e, \tilde{c} \text{ in } s\{e(x) := q : l\langle e(y) \rangle\}\]
\text{ if } s(e(x)) = q
\]

\[
\text{run } [X = A; \overrightarrow{I}e, \tilde{c} \text{ in } s \rightarrow \text{run } [\overrightarrow{I}]e, \tilde{c} \text{ in } s\{X := Ae\}
\]
\[
\text{run } [\text{new } x; \overrightarrow{I}e, \tilde{c} \text{ in } s \rightarrow \text{run } [\overrightarrow{I}]e\{x := y\}, \tilde{c} \text{ in } s\{y := \bullet\}
\text{ if } y \text{ not in } \text{dom}(s)
\]
Environment Machine—Soundness

Theorem 2 (Soundness)

1. If $E \rightarrow E'$, then $\llbracket E \rrbracket \rightarrow E'$;

2. If $\text{run } T_0 \emptyset \text{ in } \epsilon \downarrow E$ then $\llbracket E \rrbracket \not\rightarrow$. 
Byte Code Machine

Program

```
main
new %1
new %2
sw %1,0(%2)
new %3
fork2 %2,%3
...
...
fork X,%5
```

X
```
new %1
new %2
frm %3
sw %2,0(%3)
sw %1,1(%3)
new %4
fork1 %3,%4
...
...
```

t0 = { X1,...,X4 }

X1
```
new %1
...
...
```

Global Registers

Registers

Heap

\[
\begin{align*}
%hp & \quad %eq \\
%pc[0] & \quad %p[0] \\
%pc[1] & \\
%f[0] & \\
\end{align*}
\]

Thread Pool
Core Instruction Set

`frm %i,n`  Frame allocation

`new %i`  Queue allocation

`lw %i,k(%j)`  Load

`sw %i,k(%j)`  Store

`forko %i,%j`  Fork on object

`forkm %i,%j`  Fork on message

`forkd X,%i`  Fork on definition

`switch n`  Thread switch

`newt`  Load new thread
Fork Instructions

forkd

X<y>

--- running thread

--- object resource

--- message token

forko

x=M

forkm

x.1<y>
Byte Code Machine—Soundness

To be done.

Main obstacle: translation from the environment machine may need type information.
Distribution and Mobility

Lexical Scoping (Ravara’s talk)

Remote operations:

1. Apply the name translation function to the byte code;
TyCO—the Language

- Concurrent: sequential constructs are derived;
- Object-based: direct support for objects and classes;
- Implicitely typed: predicative polymorphic (ML-style) type inference system;
- Version 0.2:
  - Primitive types: integer, boolean, string, float;
  - Source split amongst different files;
  - New derived constructors;
TyCOc—the Compiler

- Generates byte-code;
- Open source;
- Uses Appel’s Tiger book ideas;
- javadoc;
- “Didactic”.
- 10,000 lines of Java code.
TyCO—the Virtual Machine

1. Interpretes byte-code;
2. Highly specialized Turner Machine;
3. Direct support for objects;
4. Competes with Pict and Oz.
5. 1,700 lines of C code.
def Account (self, balance) =

    self ? {

        deposit (amount) =

            Account [self, balance + amount]

        balance (replyto) =

            replyto ! [balance] |

            Account [self, balance]

        withdraw (amount, replyto) =

            ...

    }

in
Withdraw method

\[
\text{withdraw (amount, replyto) = if amount} \geq \text{balance then}
\]
\[
\text{replyto ! overdraft [] | Account [self, balance]}
\]
\[
\text{else}
\]
\[
\text{replyto ! dispense [] | Account [self, balance - amount]}
\]
Concurrent Objects

- procedures that generate objects are of a special form:
  \[ \text{def } X (\ldots) = \text{self ? \{} \text{methods} \} . \]

- Each method re-constructs the object if and \textit{when} needed.

- There are no (imperative) variables: object’s attributes are all updated at the same time, when the procedure recurs.

- \texttt{self} is no different from any other parameter.
Using the bank account

#include "account.tyc"

- - A bank account, located at channel b, initial balance 10

new b

Account [b, 10] |

b ! deposit [60] |

case b ! withdraw [20] of {
  overdraft () = io ! prints ["Overdraft, says Jack\n"]
  dispense () = io ! prints ["Got my 20, says Jack\n"]
}
case is derived

new replyto

b ! withdraw [20, replyto] |
replyto ? {
  overdraft () = io ! prints ["Overdraft, says Jack\n"]
  dispense () = io ! prints ["Got my 20, says Jack\n"]
}