Virtual Machines for Process Calculi

Vasco T. Vasconcelos

University of Lisbon

Joint work with

Luís Lopes and Fernando 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 mobilility.**

Outline

- **Four machines:**
  - **Threaded TyCO;**
  - **Simple Machine;**
  - **Environment Machine;**
  - **Byte Code Machine;**
- **Distribution and Mobility**
- **TyCOprogramming language, tycoc, tyco**

Threaded TyCO—Syntax

T	$::=$	$[\vec{I}]$	**Thread**
I	$::=$	$S \mid X\langle\vec{x}\rangle \mid$ **new** x	**Instruction**
S	$::=$	$x = M \mid x.l\langle\vec{x}\rangle \mid X = A$	**Store**
M	$::=$	$\{\widetilde{l = A}\}$	**Method map**
A	$::=$	$(\vec{x})T$	**Abstraction**

Threaded TyCO

Machine states.

$$\textbf{run } \tilde{T} \textbf{ in } \tilde{S}$$

Structural congruence

$$\tilde{\alpha} \equiv \tilde{\alpha}' \qquad \textbf{if } \tilde{\alpha} \textbf{ is a permutation of } \tilde{\alpha}'$$

$$[\,], \tilde{T} \equiv \tilde{T}$$

Threaded TyCO—Reduction

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

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

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

$$\textbf{run}\ [x = M; \vec{I}], \tilde{T}\ \textbf{in}\ \tilde{S} \rightarrow \textbf{run}\ [\vec{I}], \tilde{T}\ \textbf{in}\ x = M, \tilde{S}$$

$$\textbf{run}\ [x.l\langle\vec{y}\rangle, \vec{I}], \tilde{T}\ \textbf{in}\ \tilde{S} \rightarrow \textbf{run}\ [\vec{I}], \tilde{T}\ \textbf{in}\ x.l\langle\vec{y}\rangle, \tilde{S}$$

$$\textbf{run}\ [X = A; \vec{I}], \tilde{T}\ \textbf{in}\ \tilde{S} \rightarrow \textbf{run}\ [\vec{I}], \tilde{T}\ \textbf{in}\ X = A, \tilde{S}$$

$$\textbf{run}\ [\textbf{new}\ x; \vec{I}], \tilde{T}\ \textbf{in}\ \tilde{S} \rightarrow \textbf{run}\ [\{y/x\}\vec{I}], \tilde{T}\ \textbf{in}\ \tilde{S} \quad \textbf{if } y \textbf{ not free in}$$

Threaded TyCO—Soundness

Theorem 1 (Soundness) *1. If* $C \rightarrow C'$, *then either* $[\![C]\!] \equiv [\![C]\!]'$ *or* $[\![C]\!] \rightarrow [\![C]\!]'$;

2. If **run** T_0 **in** $\varepsilon \downarrow C$ *then* $[\![C]\!] \not\rightarrow$.

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** Ae**, and from object tags** x **to queues** q **of method closures** Me **or of messages contents** $l\langle\vec{x}\rangle$**.**

Environment Machine—Reduction I

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

$$\textbf{if } s(e(x)) = Me' : q, M.l = (\vec{z})T$$

$$\textbf{run } [x = M;\vec{I}]e,\tilde{c} \textbf{ in } s \longrightarrow \textbf{run } [\vec{I}]e,\tilde{c},Te\{\vec{z} := \vec{y}\} \textbf{ in } s\{e(x) := q\}$$

$$\textbf{if } s(e(x)) = l\langle\vec{y}\rangle : q, M.l = (\vec{z})T$$

$$\textbf{run } [X\langle\vec{y}\rangle;\vec{I}]e,\tilde{c} \textbf{ in } s \longrightarrow \textbf{run } [\vec{I}]e,\tilde{c},Te'\{\vec{z} := e(\vec{y})\} \textbf{ in } s$$

$$\textbf{if } s(X) = ((\vec{z})T)e'$$

Environment Machine—Reduction II

$$\textbf{run}\ [x = M; \vec{I}]e, \tilde{c}\ \textbf{in}\ s \longrightarrow \textbf{run}\ [\vec{I}]e, \tilde{c}\ \textbf{in}\ s\{e(x) := q : Me\}$$
$$\textbf{if}\ s(e(x)) = q$$

$$\textbf{run}\ [x.l\langle\vec{y}\rangle, \vec{I}]e, \tilde{c}\ \textbf{in}\ s \longrightarrow \textbf{run}\ [\vec{I}]e, \tilde{c}\ \textbf{in}\ s\{e(x) := q : l\langle e(\vec{y})\rangle\}$$
$$\textbf{if}\ s(e(x)) = q$$

$$\textbf{run}\ [X = A; \vec{I}]e, \tilde{c}\ \textbf{in}\ s \longrightarrow \textbf{run}\ [\vec{I}]e, \tilde{c}\ \textbf{in}\ s\{X := Ae\}$$

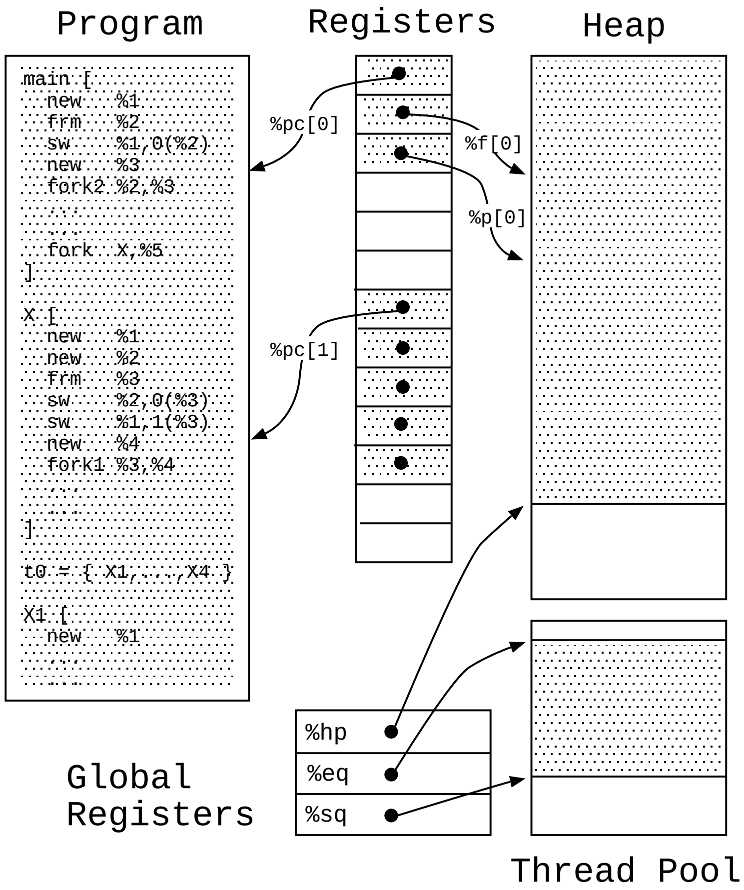
$$\textbf{run}\ [\textbf{new}\ x; \vec{I}]e, \tilde{c}\ \textbf{in}\ s \longrightarrow \textbf{run}\ [\vec{I}]e\{x := y\}, \tilde{c}\ \textbf{in}\ s\{y := \bullet\}$$
$$\textbf{if}\ y\ \textbf{not in dom}(s)$$

Environment Machine—Soundness

Theorem 2 (Soundness)

1. *If* $E \longrightarrow E'$, *then* $[\![E]\!] \longrightarrow E'$;
2. *If* **run** $T_0 \emptyset$ **in** $\varepsilon \downarrow E$ *then* $[\![E]\!] \not\longrightarrow$.

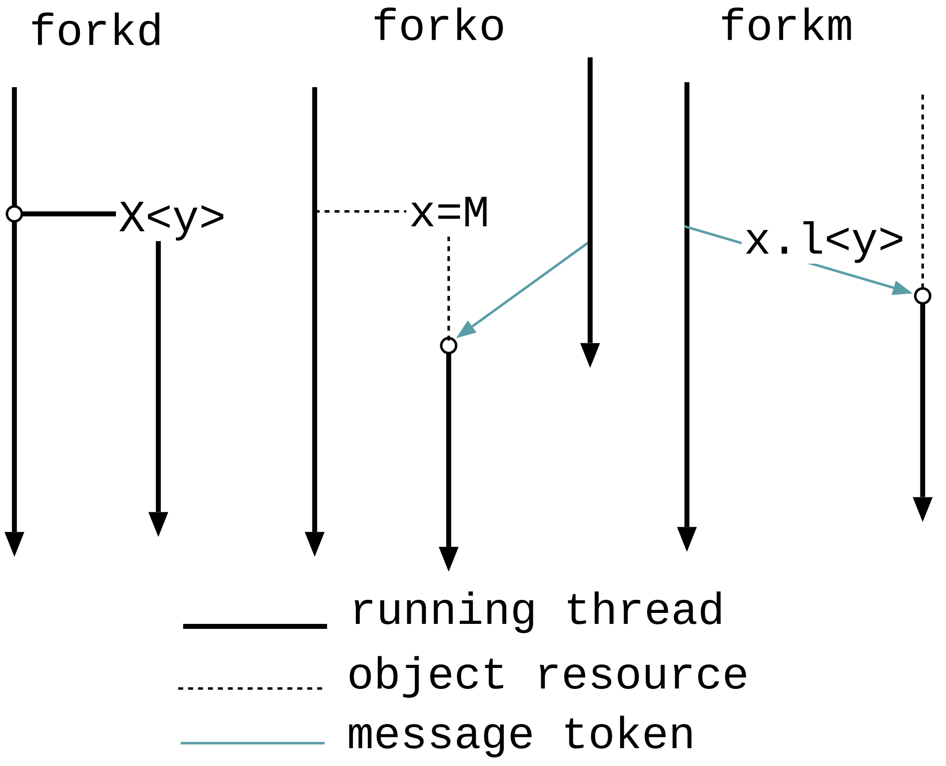
Byte Code Machine



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



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;**
2. **Marshall, unmarshall.**

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

```
withdraw (amount, replyto) =
  if amount > balance
  then
    replyto ! overdraft [] |
    Account [self, balance]
  else
    replyto ! dispense [] |
    Account [self, balance - amount]
```

Concurrent Objects

- **Procedures that generate objects are of a special form:** **def X (. . .) = self ? { *methods* }.**
- **Each method re-constructs the object if and** *when* **needed.**
- **There are no (imperative) variables: object's attributes are all updated at the same time, when the procedure recurs.**
- **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”]
}
```