Fine-Grained Multithreading with Process Calculi

Luís Lopes, Vasco T. Vasconcelos, and Fernando Silva

Abstract—This paper presents a multithreaded abstract machine for the TyCO process calculus. We argue that process calculi provide a powerful framework to reason about fine-grained parallel computations. They allow for the construction of formally verifiable systems on which to base high-level programming idioms, combined with efficient compilation schemes into multithreaded architectures.

Index Terms—Process-calculus, abstract-machine, multithreading.

1 Introduction

In recent years researchers have devoted great effort to providing semantics for pure concurrent/parallel programming languages within the realm of process-calculi. Milner et al.'s π-calculus [19] or an asynchronous formulation due to Honda and Tokoro [11] and Boudol [6] have been the starting point for most of these attempts. Several forms and extensions of the asynchronous π-calculus have since been proposed to provide for more direct programming styles and to improve efficiency and expressiveness [7], [9], [31].

Dataflow and von Neumann architectures represent the two extremes in a continuous design space. In the dataflow model, computations are triggered by the availability of all input values to an instruction (the *firing rule*). This makes the model totally asynchronous and the instructions self-scheduling. Dataflow architectures range from pure dataflow [3], [10], [23], hybrid dataflow/control-flow [8], [21], [22], and, lately, multithreaded RISC [1], [2], [25] designs. Multithreading aims to provide high processor utilization in the presence of large memory or interprocessor communication latency. High latency operations are overlapped with computation by rapidly switching to the execution of other threads.

Next generation microprocessor design coupled with VLSI technology point to multithreaded hardware as typified by IBM's Power4 and Sun's MAJC, featuring multiple RISC/VLIW cores and very high bandwidth interprocessor connections, aiming at large grain multithreading. On another scale, current superscalar microprocessor designs such as the MIPS R10k, the PA-RISC 8k, and the Alpha 21264 implement what is commonly called *micro-dataflow* using moderately large instruction windows, out-of-order execution, dynamic dispatch, and register renaming. The next generation superscalar microprocessors will require a greater amount of fine-grained parallelism to fully explore their aggressive dynamic dispatch. In this context, hardware support for multithreading can provide a solution by allowing fast context switches, overlapping memory loads, or interprocessor communication with computation. Moreover, multithreading may remove most pipeline hazards in current von Neumann implementations, thus avoiding the need for complex forwarding and branch prediction logic. However, single-thread performance in multithreaded architectures is typically low, this having a negative impact on individual application's performance. The ideal situation would call for applications themselves to be partitioned into several fine-grained threads by a compiler and the multithreading hardware would then overlap the multiple threads from that single application.

Programming paradigms and compilations techniques adequately to profit from this kind of hardware are major research areas. In recent years, the focus has shifted from pure dataflow languages, such as Id [20], to numerically oriented Sisal [17], and, finally, to the compilation of more conventional languages, such as Cilk [5] (a multithreaded C) and ML on von Neumann architectures, a shift driven by Nikhil and Arvind's seminal work on P-RISC [21].

Languages that can be efficiently compiled from high-level constructs into low-level, fine-grained, threads are quite suitable for multithreaded computing. In this context, there are several advantages in using process calculi [9], [11], [31] for concurrent and parallel computing. Process calculi provide a natural programming model as they deal with the notions of communication and concurrency. They usually have very small kernel calculi, with well-understood semantics that form ideal frameworks for language implementations. This significantly diminishes the usual gap between the semantics of the language and that of its implementation. The mappings between high level abstractions into such kernel calculi provide natural compilation schemes that expose thread parallelism at the instruction set architecture (ISA) level. Moreover, explicit communication allows the compiler to extract important hints of how and when a running thread should be suspended (typically,

- *L. Lopes and F. Silva are with the Department of Computer Science and LIACC, Faculty of Sciences, University of Porto, Rua do Campo Alegre, 823, 4150-180 Porto, Portugal. E-mail: {lblopes, fds}@ncc.up.pt.*
- *V.T. Vasconcelos is with the Deparment of Informatics, Faculty of Sciences, University of Lisbon, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: vv@di.fc.ul.pt.*

Manuscript received 1 Sept. 2000; accepted 8 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113592.

0018-9340/01/$10.00 © 2001 IEEE

when a high latency load or interprocessor communication occurs). Finally, these calculi also commonly feature type systems that not only allow checking the type safeness of programs but also collecting important information for code optimization, namely to minimize the amount of *tokens* used in the computations [4], [13], [14].

We argue that process calculi provide a powerful framework to reason about fine-grained parallel computations, allowing, for example, the construction of formally verifiable systems, high-level programming idioms, and optimizing compilers based on static type-checking information. We present a small, asynchronous, kernel object language based in a process calculus. From a programming point of view, such languages are suitable to express dataflow computations and, in particular, multithreaded computations. The base process calculus is quite expressive and many interesting programming idioms can be encoded in its kernel form. These encodings, on the other hand, provide a straightforward way of compiling high-level programs into low-level, threaded code, thus exposed to the ISA level. The asynchronous character of the calculus makes the interleaved/parallel execution of threads a natural choice. To formalize these ideas, we introduce a specification for a multithreaded abstract machine that grows from Turner's abstract machine for the asynchronous π-calculus [27] and which possesses important runtime properties, namely its soundness with respect to the base calculus and the absence of deadlocks in well-typed programs.

The remainder of the paper is organized as follows: Section 2 presents the Threaded TyCO calculus. Sections 3 and 4 present two multithreaded abstract machines for the calculus. Sections 5 and 6 present some properties of both machines. Section 7 describes an implementation for the second abstract machine, namely the fundamental data structures and the instruction set architecture. Section 8 describes a compilation scheme for languages, using the calculus as an intermediate representation language. Finally, Section 9 compares this approach to other related work and issues some conclusions and future research directions.

2 The Threaded TyCO Calculus

Typed Concurrent Objects, TyCO for short, is a form of the asynchronous π-calculus featuring first class objects, asynchronous messages, and polymorphic process definitions [29], [31]. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. This section recasts TyCO with a multithreaded dataflow flavor, whose main abstractions are *threads*, *tokens*, and *resources*. We call the new calculus TTyCO.

Below, for each syntactic category α, we let $\tilde{\alpha}$ denote the *collection* $\alpha_1, \ldots, \alpha_n$ and let $\vec{\alpha}$ denote the *sequence* $\alpha_1; \ldots; \alpha_n$, for $n \geq 0$. When n is 0, $\tilde{\alpha}$ is represented by ε and $\vec{\alpha}$ by a space (' '). The difference between the collection $\tilde{\alpha}$ and the sequence $\vec{\alpha}$ is that we allow permutations on the former, but not on the latter.[2]

Threads T are sequential compositions of atomic instructions. The syntax $[\vec{I}]$ denotes a thread composed of a sequence of instructions $\vec{I}$. Concurrent composition of threads is denoted by $\tilde{T}$. Threads constitute the unit of concurrency in the calculus.

Instructions I are the constituents of threads and may generate tokens or resources. Resources are pieces of code whose execution is pending on the arrival of a token. A resource may be an object $x = M$ or a polymorphic thread definition $X = A$. In each case, x or X is the tag of the resource. We call x-tags *object tags* and X-tags *thread tags*. Tokens are pieces of data that activate resources, ultimately producing new threads. Tokens come in two flavors: asynchronous messages $x.l\langle\vec{y}\rangle$ that invoke the method l in an object resource tagged with x and an instance creation $X\langle\vec{x}\rangle$ that produces a copy of the thread tagged with X. A final instruction, **new** x, allows the creation of a new tag in a given thread.

Thread abstractions A and *method maps* M complete the calculus. A thread abstraction $(\vec{x})T$ is simply a thread abstracted on a sequence of object tags. Think of these tags as the parameters of the thread; the token that activates a resource $X = A$ also provides the arguments for the thread. Methods form the bodies of objects. They are maps (that is, partial functions of finite domain) from labels into thread abstractions. This time, the token that activates a resource $x = M$ must supply a label (to obtain an abstraction) and the arguments (to obtain a runnable thread).

Given infinite disjoint sets for object tags (x, y, z), for thread tags (X), and for labels (l), we write the full syntax of the calculus as follows:

$$\begin{array}{rcll}
T & ::= & [\vec{I}] & \text{Thread} \\
I & ::= & S \mid X\langle\vec{x}\rangle \mid \mathbf{new}\ x & \text{Instruction} \\
S & ::= & x = M \mid x.l\langle\vec{x}\rangle \mid X = A & \text{Store} \\
M & ::= & \{\widetilde{l = A}\} & \text{Method map} \\
A & ::= & (\vec{x})T & \text{Abstraction.}
\end{array}$$

We assume that the names in the sequence $\vec{x}$ of an abstraction $(\vec{x})T$ are pairwise distinct. Also, in a sequence of instructions $X_1 = A_1; \ldots; X_n = A_n$, we assume that tags X_i are pairwise distinct. The reason why we identify a separate category for store items S becomes clearer in the next section as we describe an abstract machine to run TTyCO threads.

To illustrate the syntax and main components of the calculus, we sketch a small example of a polymorphic buffer cell.

```
[
  Cell = (self, value) [
    self = {
      read = (reply)[ reply.val⟨value⟩; Cell⟨self,value⟩ ],
      write = (newval)[ Cell⟨self, newval⟩ ]
    }
  ];
  new intcell; Cell⟨intcell, 5⟩;
  new boolcell; Cell⟨boolcell, false⟩;
  intcell.write⟨7⟩;
  new r; boolcell.read⟨r⟩;
```

[1] More precisely, $(\alpha, ';', '')$ is a *commutative monoid*, whereas $(\alpha, ',', \varepsilon)$ is a *commutative monoid*.

```
  r = {val = (x)[io.print⟨x⟩]}
]
```

In the example, we define a thread abstraction, the Cell, with two attributes: the location self and the value value of the cell. The thread body is defined as a single object with two methods, one for reading the current cell value and another to change it. The recursion at the end of each method keeps the cell alive after a reduction. TTyCO inherits from TyCO a predicative polymorphic type-system and, as a result, the Cell abstraction is polymorphic on the attribute value. Next, we create two instances of Cell: one with an integer attribute, the other with a Boolean attribute. Finally, we write the value 7 in the intcell (originally with the value 5); read the value from boolcell and wait for a reply in tag r. When the reply message arrives, a new thread is spawned that prints the value in the message.

3 A Simple Abstract Machine

This section presents an abstract machine for interpreting TTyCO programs. The machine evolves by maintaining a store of resources and tokens and analyzing the instructions in each thread, from left to right. Resources for which there are no matching tokens pending are moved to the store; tokens for which there are no available resources are moved to the store.

3.1 Machine States

A tag x is *bound* in the sequence of instructions $\vec{I'}$ in a thread of the form $[\vec{I}; \mathbf{new}\, x; \vec{I'}]$ and in the thread T of an abstraction $(\vec{y}x\vec{z})T$; otherwise it is *free*. For tags denoting thread definitions, the rule is slightly different. Given a thread of the form $[\vec{I}; X_1 = A_1; \ldots; X_n = A_n; \vec{I'}]$, each tag X_i is *bound* in $A_1, \ldots, A_n$ and in $\vec{I'}$; otherwise, it is *free*.

A direct consequence of this definition is that, in a closed program, when scanning the instructions in a thread from left to right, a resource $X = A$ always appears prior to a token $X\langle\vec{x}\rangle$. On the other hand, since the creation of a new tag x is decoupled from the creation of its resources, we cannot guarantee that a resource $x = M$ appears prior to a token $x.l\langle\vec{y}\rangle$. As such, instance creation tokens $X\langle\vec{x}\rangle$ will never find their way into the store, whereas message tokens $x.l\langle\vec{y}\rangle$ may.

Our machine evolves by rewriting states into states. A state of the machine is represented by a term

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

denoting a pool of threads $\tilde{T}$ running on a pool of available resources and tokens $\tilde{S}$. We use the letter C to range over states. Items in the store are a subset of the possible instructions; this is the reason why we have anticipated a separate subcategory S of instructions in the syntax of the calculus.

We take the view that programs are closed for tags, that is, the initial thread contains no free tags. We also assume that all bound tags (object and thread) in the initial thread are pairwise distinct. Given such a thread T_0 we build the *initial state* of the machine as

$$\mathbf{run}\ T_0\ \mathbf{in}\ \varepsilon.$$

3.2 Structural Congruence

Following Milner [18], we divide the computational rules of the calculus into two parts: the *structural congruence* rules and the *reduction* rules. Structural congruence rules allow the rewriting of terms until they are in the form required by reduction, thus simplifying the presentation of the latter.

For α, a thread T, or a store item S, the structural congruence relation is the smallest congruence relation on states that include the following rules:

$$\begin{aligned} \tilde{\alpha} &\equiv \tilde{\alpha}' \quad \text{if } \tilde{\alpha} \text{ is a permutation of } \tilde{\alpha}' \\ [], \tilde{T} &\equiv \tilde{T}. \end{aligned}$$

Notice that concurrency and nondeterminism are introduced by allowing arbitrary permutations among threads and items in the store. The second rule allows the garbage collection of an empty thread.

3.3 Reduction

Given the notion of free tags described in Section 3.1, we denote by $\{\vec{x}/\vec{y}\}T$ the usual (capture-avoiding) substitution of $\vec{x}$ for $\vec{y}$ in T. To extract the abstraction associated with label l in a method map M, we use the notation $M.l$ rather than the conventional notation for maps $M(l)$. To apply an abstraction to a sequence of object tags, we write $((\vec{y})T)\langle\vec{x}\rangle \stackrel{\text{def}}{=} \{\vec{x}/\vec{y}\}T$ so that, when M is $\{l = (\vec{y})T, \ldots\}$, the expression $M.l\langle\vec{x}\rangle$ stands for the thread $\{\vec{x}/\vec{y}\}T$.

Computation is driven by the interaction between concurrent threads of execution. Each thread produces new tags, tokens, and resources that interact with those already in the store. New threads result from appropriate matching between tags in the threads and in the store. Fig. 1 summarizes the rules. We denote by $\rightarrow^*$ the reflexive and transitive closure of the $\rightarrow$ relation.

FORK-M: A method invocation $x.l\langle\vec{y}\rangle$ at the head of a thread selects the method l of an object $x = M$ in the store. The matching is done via the name tag, here x. The result is a new thread whose parameters have been replaced by the arguments $\vec{y}$. FORK-O: Inversely, an object $x = M$ is triggered by a message $x.l\langle\vec{y}\rangle$ waiting in the store. In each case, the messages and objects are consumed. These two forms of reduction are called *communication*.

FORK-D: Another form of reduction occurs when a new instance $X\langle\vec{x}\rangle$ at the head of a thread meets a definition $X = A$ in the store. The result is a thread whose parameters have been replaced by the arguments $\vec{x}$. Notice that the resource is not consumed in the process; we would not have unbounded computations otherwise. This form of reduction is called *instantiation*.

STORE-O, STORE-M, STORE-D: When there is no match in the store for a token or a resource appearing in a thread, these are put into the store. The STORE rules must only be tried after the FORK rules.

NEW: To create a new (object) tag, we rely on the set of tags being infinite and pick one unused in the rest of the state. Finally, rule STRUCT brings structural congruence into reduction.

Analyzing the rules, we see that the machine halts when the pool of threads empties. *Final states* are of the form

$$\begin{array}{lr}
\textsf{run}\ [x.l\langle\vec{y}\rangle;\vec{I}],\tilde{T}\ \textsf{in}\ x=M,\tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T},M.l\langle\vec{y}\rangle\ \textsf{in}\ \tilde{S} & (\textsc{fork-m})\\
\textsf{run}\ [x=M;\vec{I}],\tilde{T}\ \textsf{in}\ x.l\langle\vec{y}\rangle,\tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T},M.l\langle\vec{y}\rangle\ \textsf{in}\ \tilde{S} & (\textsc{fork-o})\\
\textsf{run}\ [X\langle\tilde{y}\rangle;\vec{I}],\tilde{T}\ \textsf{in}\ X=A,\tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T},A\langle\vec{y}\rangle\ \textsf{in}\ X=A,\tilde{S} & (\textsc{fork-d})\\
\textsf{run}\ [x=M;\vec{I}],\tilde{T}\ \textsf{in}\ \tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T}\ \textsf{in}\ x=M,\tilde{S} & (\textsc{store-o})\\
\textsf{run}\ [x.l\langle\vec{y}\rangle,\vec{I}],\tilde{T}\ \textsf{in}\ \tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T}\ \textsf{in}\ x.l\langle\vec{y}\rangle,\tilde{S} & (\textsc{store-m})\\
\textsf{run}\ [X=A;\vec{I}],\tilde{T}\ \textsf{in}\ \tilde{S}\rightarrow\textsf{run}\ [\vec{I}],\tilde{T}\ \textsf{in}\ X=A,\tilde{S} & (\textsc{store-d})\\
\textsf{run}\ [\textsf{new}\ x;\vec{I}],\tilde{T}\ \textsf{in}\ \tilde{S}\rightarrow\textsf{run}\ [\{y/x\}\vec{I}],\tilde{T}\ \textsf{in}\ \tilde{S}\quad\text{if } y \text{ not free in } \vec{I},\tilde{T},\tilde{S} & (\textsc{new})\\
\dfrac{C\equiv C'\qquad C'\rightarrow C''\qquad C''\equiv C'''}{C\rightarrow C'''} & (\textsc{struct})
\end{array}$$

Fig. 1. The simple machine for TTyCO.

$$\mathbf{run}\ \varepsilon\ \ \mathbf{in}\ \tilde{S}.$$

We illustrate the dynamics of the simple machine by executing the Cell example described in Section 2. Suppose that the initial thread is run to the end. After nine reduction steps (there are nine instructions in the initial thread), we get the state

run [i={i/self,5/value}M], [b={b/self,false/value}M]
in Cell=A, i.write⟨7⟩, b.read⟨r'⟩

for appropriate M and A taken from the example. We have performed the substitutions {i/intcell}, {b/boolcell}, and {r'/r} for three **new** rules. Next, we may run the two threads to the end, to obtain

run [Cell⟨i,7⟩], [r'.val⟨false⟩; Cell⟨b,false⟩]
in Cell=A, r'= {val = (x)[io.print⟨x⟩]}

Running the two new threads to the end, we get

run [i={i/self,7/value}M], [io.print⟨false⟩],
[b={b/self,false/value}M]
in Cell=A, i.write⟨7⟩, b.read⟨r'⟩.

Finally, assuming that print forks no thread, the machine halts at the state

run ε
in Cell=A,i={i/self,7/value}M,b={b/self,false/value}M

with two cell objects and the cell thread definition in the store.

3.4 Discussion

With respect to TyCO, we have traded the explicit parallel composition of processes for the sequential nature of threads, coupled with the nondeterminism in the selection of a resource or a token (captured by the structural congruence relation). Threaded TyCO is sound with respect to TyCO; see Section 4 for details.

We have not said what happens to the machine when, in the FORK rules, thread instantiation or method selection is not defined. In fact, rule FORK-D can only fire if $A\langle\vec{y}\rangle$ is defined, that is, when the lengths of $\vec{z}$ and $\vec{y}$ match, for A of the form $(\vec{z})T$. Rules FORK-M and FORK-O can only fire if l is in the domain of M and $A\langle\vec{y}\rangle$ is defined, for $M(l)=A$. Fortunately, we can statically guarantee that such runtime errors do not occur for a certain class of programs, exactly those programs that are typable in a (decidable) type system [28], [31]. For such programs, the abstract machine does not deadlock, that is, it either halts or runs indefinitely [15].

To account for multithreading, we allow permutations on threads via STRUCT rule. Notice that, at any point in the execution of a machine, the running thread (the one at the "left" of the pool) may be switched, by starting the next reduction step with rule STRUCT (where the $C\equiv C'$ part switches the order of threads as desired). Such behavior can be described by a rule of the form

$$\mathbf{run}\ \tilde{T_1},T,\tilde{T_2}\ \mathbf{in}\ s\rightarrow\mathbf{run}\ T,\tilde{T_1},\tilde{T_2}\ \mathbf{in}\ s.$$

Permutation on threads precludes the fairness of the machine in the sense that any given instruction in any given thread always gets executed, contrary to the case with the machine proposed by Turner [27].

4 The Simple Machine Is Sound

This section presents the TyCO process calculus, a proof that the simple machine presented in Section 3 is sound with respect to TyCO, and a discussion on the incompleteness of the machine. The soundness result is important since it tells us that the simple machine can be fully simulated in TyCO (and, therefore, in the asynchronous π-calculus), a crucial step in the development of a robust machine specification.

4.1 Typed Concurrent Objects

The TyCO process calculus is quite similar to its threaded version presented in Section 2. The main difference is that threads are replaced by the parallel composition of processes and that mutually recursive process definitions are explicit.

Given infinite disjoint sets for names (x,y,z), for process variables (X), and for labels (l), we write the full syntax of TyCO as follows:

$$\boxed{\begin{array}{rl}
(P, `\mid', \mathbf{0}) \text{ forms a commutative monoid} & \\
\textbf{new } x\ \mathbf{0} \equiv \mathbf{0}, & \\
\textbf{new } x \textbf{ new } y\ P \equiv \textbf{new } y \textbf{ new } x\ P, & \\
\textbf{new } x\ P \mid Q \equiv P \mid \textbf{new } x\ Q & \text{if } x \notin \text{fn}(P) \\
\textbf{def } D \textbf{ in } \mathbf{0} \equiv \mathbf{0} & \\
\textbf{def } D \textbf{ in new } x\ P \equiv \textbf{new } x \textbf{ def } D \textbf{ in } P & \text{if } x \notin \text{fn}(D) \\
\textbf{def } D \textbf{ in } P \mid Q \equiv P \mid \textbf{def } D \textbf{ in } Q & \text{if } \text{dom}(D) \cap \text{fv}(P) = \emptyset \\
\textbf{def } D \textbf{ in def } D' \textbf{ in } P \equiv \textbf{def } D, D' \textbf{ in } P & \text{if } \text{dom}(D) \cap \text{dom}(D') = \emptyset \\
\alpha \equiv \alpha' & \text{if } \alpha \text{ is a permutation of } \alpha'
\end{array}}$$

Fig. 2. Structural congruence.

$$\begin{array}{rcll}
P & ::= & \mathbf{0} \mid P \mid P' \mid \textbf{new } x\ P & \text{Processes} \\
 & & x = M \mid x.l\langle\vec{x}\rangle & \\
 & & \textbf{def } D \textbf{ in } P \mid X\langle\vec{x}\rangle & \\
D & ::= & \widetilde{X = A} & \text{Definitions} \\
M & ::= & \{\widetilde{l = A}\} & \text{Method Maps} \\
A & ::= & (\vec{x})P & \text{Abstractions.}
\end{array}$$

Definitions D and method maps M are to be understood as maps from process variables (respectively, labels) into abstractions. The intended meaning of each constructor should be evident from the discussion on TTyCO in Section 2. The scope of the name introduced by **new** extends as far to the right as possible in the same ways as the scope of x in a thread $[\vec{I}; \textbf{new}\ x; \vec{I'}]$ encompasses the whole $\vec{I'}$. Also, the scope of the process variables introduced by **def** extends as far to the right as possible.

To be in line with the usual conventions in process calculi, we have shifted terminology: we now call *names* to *object tags*, and *process variables* to *thread tags*.

Contrary to TTyCO, processes need not be closed for names and process variables. Name x is *bound* in the process P of an abstraction $(\vec{y}x\vec{z})P$ or a scope restriction **new** $x\ P$; it is *free* otherwise. For each syntactic category α, the set $\text{fn}(\alpha)$ of the *free names* in an α is defined accordingly and so is the usual (capture-free) substitution of $\vec{x}$ for $\vec{y}$ in P, denoted $\{\vec{x}/\vec{y}\}P$. Given a process of the form **def** $X_1 = A_1, \ldots, X_n = A_n$ **in** P, each tag X_i is *bound* in the abstractions $A_1, \ldots, A_n$ and in the process P; it is *free* otherwise. The set $\text{fv}(P)$ of the *free (process) variables* in a process P is defined accordingly. *Alpha conversion*, both for names and for process variables, is defined in the usual way.

Similarly to TTyCO, the operational semantics is given by a reduction relation built on top of a structural congruence. Here, structural congruence, also denoted $\equiv$, is somewhat more complex than in TTyCO. It is defined as the smallest congruence relation over processes that includes alpha conversion and is induced by the rules in Fig. 2, where α is a method map M or a definition D.

The first rule allows garbage collecting **0** processes and states the commutativity and associativity of the parallel composition operator. The next three **new**-related rules allow garbage collecting names with empty scopes, commuting **new** statements, and extending the scope of **new** over parallel composition, provided there is no capture of names. Then, the four **def**-related rules account for the garbage collection of definitions with empty bodies, for the commutation of **new** and **def**, for the extension of the scope of **def** over parallel composition, and for the coalescence of process declarations, all avoiding the capture of free names and free process variables.

The rules that define reduction are gathered in Fig. 3. The dynamics of the two axioms, COMM and INST, can be understood from that of rules FORK-M/FORK-O and FORK-D of the simple machine for TTyCO (Section 3.3), respectively. The next three rules allow reduction to happen underneath parallel composition, name restriction, and definitions, respectively. Finally, rule STRUCT brings structural congruence into reduction.

4.2 Soundness

We translate states of the form

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

where $\tilde{S}$ contains only objects $x = M$ or messages $x.l\langle\vec{x}\rangle$.[2] It should be easy to see that any state can be converted into a structural equivalent of the above form. As such, in the sequel, we reassign letter S to denote an object or a message. The translation of machine states into TyCO processes is as follows:

$$\begin{array}{l}
[|\textbf{run } \tilde{T} \textbf{ in } D, \tilde{S}|] \overset{\text{def}}{=} \\
\textbf{new } x_1 \ldots \textbf{ new } x_n \textbf{ def } [|D|] \textbf{ in } [|\tilde{S}|] \mid [|\tilde{T}|],
\end{array}$$

where $x_1, \ldots x_n$ are the free names in the state, that is, in $\tilde{T}, D, \tilde{S}$.

The maps $[|D|]$ and $[|\tilde{S}|]$ are the homomorphic extensions (to the respective monoids) of the map defined by the rules

$$\begin{array}{c}
[|X = A|] \overset{\text{def}}{=} X = [|A|] \\
[|x = M|] \overset{\text{def}}{=} x = [|M|] \\
[|x.l\langle\vec{x}\rangle|] \overset{\text{def}}{=} x.l\langle\vec{x}\rangle \\
[|(\vec{x})T|] \overset{\text{def}}{=} (\vec{x})[|T|] \\
[|\{\widetilde{l = A}\}|] \overset{\text{def}}{=} \{\widetilde{l = [|A|]}\}.
\end{array}$$

The map $[|\tilde{T}|]$ is the homomorphic extension of the map defined by the following rules:

[2] Abusing the notation, letter D here denotes a collection of definitions $\widetilde{X = A}$ in the simple machine.

$$\begin{array}{lr}
x = M \mid x.l\langle\vec{y}\rangle \to M.l\langle\vec{y}\rangle & (\textsc{comm}) \\
\textsf{def}\ X = A, D\ \textsf{in}\ X\langle\tilde{y}\rangle \mid P \to \textsf{def}\ X = A, D\ \textsf{in}\ A\langle\tilde{y}\rangle \mid P & (\textsc{inst}) \\
\dfrac{P \to P'}{P \mid Q \to P' \mid Q} & (\textsc{conc}) \\
\dfrac{P \to P'}{\textsf{new}\ x\ P \to \textsf{new}\ x\ P'} & (\textsc{new}) \\
\dfrac{P \to P'}{\textsf{def}\ D\ \textsf{in}\ P \to \textsf{def}\ D\ \textsf{in}\ P'} & (\textsc{def}) \\
\dfrac{P \equiv P' \qquad P' \to P''}{P \to P''} & (\textsc{struct})
\end{array}$$

Fig. 3. Reduction in TyCO.

$$\begin{aligned}
[|[]|] &\overset{\text{def}}{=} \mathbf{0} \\
[|[x.l\langle\vec{x}\rangle; \vec{I}]|] &\overset{\text{def}}{=} x.l\langle\vec{x}\rangle \mid [|\vec{I}|] \\
[|[x = M; \vec{I}]|] &\overset{\text{def}}{=} x = [|M|] \mid [|\vec{I}|] \\
[|[X\langle\vec{x}\rangle; \vec{I}]|] &\overset{\text{def}}{=} X\langle\vec{x}\rangle \mid [|\vec{I}|] \\
[|[\mathbf{new}\ x; \vec{I}]|] &\overset{\text{def}}{=} \mathbf{new}\ x\ [|\vec{I}|] \\
[|[D; \vec{I}]|] &\overset{\text{def}}{=} \mathbf{def}\ [|D \in [|\vec{I}|],
\end{aligned}$$

where, in the rule for $[D; \vec{I}]$, we assume $\vec{I}$ does not start with an instruction of the form $X = A$.

For example, the state

$$\mathbf{run}[x = \{l = (y)[]\}; x.l\langle 1\rangle]\ \mathbf{in}\ x.l\langle 2\rangle \qquad (1)$$

is translated into a process structural congruent to

$$\mathbf{new}\ x\ x.l\langle 2\rangle \mid x = \{l = (y)\mathbf{0}\} \mid x.l\langle 1\rangle. \qquad (2)$$

We say that C *converges to* C' and denote $C \downarrow C'$ if $C \to^* C'$ and $C' \not\to$. Then, the main result of this section may be presented as follows:

Theorem 1 (Soundness).

1. *If $C \to C'$, then either $[|C|] \equiv [|C|]'$ or $[|C|] \to [|C|]'$;*
2. *If $\mathbf{run}\ T_0\ \mathbf{in}\ \varepsilon \downarrow C$, then $[|C|] \not\to$.*

For the proof of the second clause, we need an auxiliary result, namely, the simple machine preserves the invariant that the store contains no redex.

Lemma 2 (Invariant). *Call a pair of the form $x = M, x.l\langle\vec{x}\rangle$ a redex.*

1. *If $\tilde{S}$ contains no redex and*

$$\mathbf{run}\ \tilde{T}\ \mathbf{in}\ \tilde{S} \to \mathbf{run}\ \tilde{T}'\ \mathbf{in}\ \tilde{S}',$$

then $\tilde{S}'$ contains no redex.

2. *If $\mathbf{run}\ T_0\ \mathbf{in}\ \varepsilon \to^* \mathbf{run}\ \tilde{T}\ \mathbf{in}\ \tilde{S}$, then $\tilde{S}$ contains no redex.*

Proof.

1. A simple induction on the derivation of a reduction step.
2. A simple induction on the length of derivation, noticing that the initial state $\mathbf{run}\ T_0\ \mathbf{in}\ \varepsilon$ is in the conditions of the first clause. □

We are now in a position to prove the main result.

Proof of Theorem 1.

1. By induction on the derivation of a reduction step. An outline follows. For the axioms, the table below summarizes the relation between $[|C|]$ and $[|C'|]$.

Axiom	Relation
FORK	$\to$
STORE	$\equiv$
NEW	$\equiv$

For the induction step, rule STRUCT, notice that $C \equiv C'$ implies $[|C|] \equiv [|C'|]$ since the maps $[|\tilde{T}|]$ and $[|\tilde{S}|]$ are homomorphisms and $[], \tilde{T} \equiv \tilde{T}$ matches $\mathbf{0}, P \equiv P$.

2. Since $C \not\to$, we know that C is of the form $\mathbf{run}\ \varepsilon\ \mathbf{in}\ D, \tilde{S}$, hence $[|C|]$ is

$$\mathbf{new}\ \tilde{x}\ \mathbf{def}\ [|D|]\ \mathbf{in}\ [|\tilde{S}|].$$

The result follows from the invariant 2.2. □

4.3 Incompleteness

The map $[|C|]$ from machine states into TyCO processes described in the previous section is not complete. In fact, there are reductions of processes $[|C|]$ that cannot be mimicked by the simple machine. One such process is described by (2) in Section 4.2: $[|C|]$ may reduce to $\mathbf{new}\ x\ x.l\langle 2\rangle$, but, due to the fact that STORE rules are always tried after FORK rules, C ((1), same section) reduces to $\mathbf{run}\ \varepsilon\ \mathbf{in}\ x.l\langle 1\rangle$ that translates into $\mathbf{new}\ x\ x.l\langle 1\rangle$.

Further, nonhalted states may be translated into halted processes. Take for C the state $\mathbf{run}\ x.l\langle\rangle\ \mathbf{in}\ \varepsilon$. Then, C reduces (to $\mathbf{run}\ \varepsilon\ \mathbf{in}\ x.l\langle\rangle$), but $[|C|] \overset{\text{def}}{=} \mathbf{new}\ x\ x.l\langle\rangle$ does not. The rule employed in the reduction is STORE-M. In general, it should be the case that, if $[|C|]$ is halted, then C converges by using rules other than the FORK rules.

4.4 An Alternative Machine

A variant of the simple machine is obtained by eliminating the rule

STORE rules must be tried only after FORK rules.

We conjecture that such a machine is complete in some particular sense. For example, the state described by (1) in Section 4.2 now reduces in two steps (using rules STORE-O and FORK-M) to $\mathbf{run}\ []\ \mathbf{in}\ x.l\langle 2\rangle$, as required.

$$
\begin{array}{rll}
\textsf{run}\ [x.l\langle\vec{y}\rangle;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c},Te'\{\vec{z}:=e(\vec{y})\}\ \textsf{in}\ s\{e(x):=q\} & \\
 & \text{if } s(e(x)) = Me' : q \quad \text{and } M.l = (\vec{z})T & (\textsc{fork-m}) \\
\textsf{run}\ [x=M;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c},Te\{\vec{z}:=\vec{y}\}\ \textsf{in}\ s\{e(x):=q\} & \\
 & \text{if } s(e(x)) = l\langle\vec{y}\rangle : q \quad \text{and } M.l = (\vec{z})T & (\textsc{fork-o}) \\
\textsf{run}\ [X\langle\vec{y}\rangle;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c},Te'\{\vec{z}:=e(\vec{y})\}\ \textsf{in}\ s & \\
 & \text{if } s(X) = ((\vec{z})T)e' & (\textsc{fork-d}) \\
\textsf{run}\ [x=M;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c}\ \textsf{in}\ s\{e(x):=q:Me\} & \\
 & \text{if } s(e(x)) = q & (\textsc{store-o}) \\
\textsf{run}\ [x.l\langle\vec{y}\rangle,\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c}\ \textsf{in}\ s\{e(x):=q:l\langle e(\vec{y})\rangle\} & \\
 & \text{if } s(e(x)) = q & (\textsc{store-m}) \\
\textsf{run}\ [X=A;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e,\tilde{c}\ \textsf{in}\ s\{X:=Ae\} & (\textsc{store-d}) \\
\textsf{run}\ [\textsf{new}\ x;\vec{I}]e,\tilde{c}\ \textsf{in}\ s \to & \textsf{run}\ [\vec{I}]e\{x:=y\},\tilde{c}\ \textsf{in}\ s\{y:=\bullet\} & \\
 & \text{if } y \text{ not in dom}(s) & (\textsc{new}) \\
\multicolumn{2}{c}{\dfrac{E \equiv E' \qquad E' \to E'' \qquad E'' \to E'''}{E \to E'''}} & (\textsc{struct})
\end{array}
$$

Fig. 4. The environment machine for TTyCO.

Also, this small change in the machine invalidates the invariant (Lemma 2) as redexes may now find their way into the store.

5 The Environment Machine

The machine proposed in Section 3 is still far from an efficient interpreter: Substitutions are actually performed by visiting the threads, the store is completely unstructured, and the machine relies on the structural congruence relation to bring forward the appropriate resource or token for a given rule. This section presents another abstract machine that avoids substitution and allows direct access to the resources and tokens in the store.

5.1 States

An *environment* e is a map from object tags into object tags. A *thread closure* c or Te is a pair composed of a thread T and an environment e. We also need closures for resources since these may go into the store. *Abstraction closures* Ae and *method closures* Me are defined as for thread closures. We evaluate closures αe such that the free tags in α are in the domain of e.

The *store*, formerly a bag of resources and messages, is given some structure, becoming a map s from thread tags X into abstraction closures Ae and from object tags x into queues q of method closures Me or of messages contents $l\langle\vec{x}\rangle$. A queue q can be regarded as a possibly empty list $\alpha_1 : \ldots : \alpha_n$; the empty queue being denoted by $\bullet$. Queues are not needed for thread definitions since the bindings in the calculus (see Section 3) guarantee that there is at most a definition $X = A$ per thread tag X. Also, messages need no accompanying environment since, when storing, we apply the current environment.

The *state* of the environment machine is represented by a term

$$\textbf{run}\ \tilde{c}\ \textbf{in}\ s,$$

denoting a pool of thread closures $\tilde{c}$ running on a map s from tags to the available resources and tokens. We use letter E to range over environment states. Given a renamed program T_0 we build the *initial state* of the machine as below, where $\emptyset$ denotes the empty environment:

$$\textbf{run}\ T_0\emptyset\ \textbf{in}\ \varepsilon.$$

5.2 Structural Congruence

Structural congruence is defined similarly to that of the simple machine (Section 3.2). The difference is that it now works only on the pool of threads. The structural congruence relation is the smallest congruence relation on states that includes the following rules:

$$
\begin{array}{rl}
\tilde{c} \equiv \tilde{c}' & \text{if } \tilde{c} \text{ is a permutation of } \tilde{c}' \\
[]e,\tilde{c} \equiv \tilde{c}. &
\end{array}
$$

5.3 Reduction

For a given map α from elements β into elements γ, we use the notation $\alpha\{\beta := \gamma\}$ for the map α' such that $\alpha'(\beta')$ is γ when β' is β and is $\alpha(\beta')$ otherwise. Fig. 4 summarizes the reduction rules for the environment machine. A synopsis of the rules follows.

FORK-M: Given a message $x.l\langle\vec{y}\rangle$ in an environment e, we look for a method closure Me' at the head of the queue associated with $e(x)$. Me' is dequeued and a new thread closure is created with the appropriate environment. This environment is the environment of M everywhere, except at the parameters $\vec{z}$ of the thread abstraction $M.l$ that are mapped into the arguments $e(\tilde{y})$.

FORK-O: Inversely, given an object $x = M$ in an environment e, we look for a message $l\langle\vec{y}\rangle$ at the head of the queue associated with $e(x)$. A new thread closure is created with the appropriate environment; $l\langle\vec{y}\rangle$ is dequeued. FORK-D is similar to FORK-M except that the abstraction closure is not removed from the store.

STORE-O, STORE-M, STORE-D: These rules behave similarly to their counterparts in the simple machine, the difference being that tokens and resources are now stored at the tail of the queue associated with the tag. Also, object tags need to be dereferenced through the environment. As with the simple machine, the STORE rules must only be tried after the FORK rules.

NEW: Instead of creating a new tag y and substituting throughout the remaining thread $\vec{I}$, we place a new binding $\{x := y\}$ in the environment and a new entry $\{y := \bullet\}$ in the store (remember that $\bullet$ denotes the empty queue).

STRUCT: As in the corresponding rule for the simple machine (Section 3.3).

5.4 Discussion

Because we have imposed a discipline (FIFO) in the access to the resources in the store, the environment machine, albeit sound, is not complete with respect to the simple machine of Section 3 (hence, with respect to TyCO); see Section 6 for details.

Also, given that FORK rules are to be tried before STORE rules, the environment machine preserves the invariant that, at any time during a computation, the queues in the store are either empty or have only messages or have only method closures [27]. This is the counterpart of the invariant (Lemma 2) of the simple machine.

6 The Environment Machine Is Sound

This section presents a proof that the environment machine of Section 5 is sound, but not complete, with respect to the simple machine of Section 3.

We start by encoding a state of the environment machine $\mathbf{run}\ \tilde{c}\ \mathbf{in}\ s$ into a state of the simple machine $\mathbf{run}\ \tilde{T}\ \mathbf{in}\ \tilde{S}$. For α a thread T, an abstraction A, or a method map M, denote by $e\alpha$ the result of the application of the substitution e to α. The translation of environment machine states into simple machine states is as follows:

$$\mathbf{run}\ \tilde{c}\ \mathbf{in}\ s \overset{\text{def}}{=} \mathbf{run}\ [|\tilde{c}|]\ \mathbf{in}\ [|s|].$$

The map $[|\tilde{c}|]$ is the homomorphic extension, within the appropriate monoids, of the map defined by the rule

$$[|Te|] \overset{\text{def}}{=} eT.$$

When the domain of s is $\alpha_1, \dots, \alpha_n$, the map $[|s|]$ is defined as follows:

$$\begin{aligned}
[|s|] &\overset{\text{def}}{=} [|\alpha_1 = s(\alpha_1)|], \dots, [|\alpha_n = s(\alpha_n)|]\\
[|X = Ae|] &\overset{\text{def}}{=} X = eA\\
[|y = \bullet|] &\overset{\text{def}}{=} \varepsilon\\
[|x = l\langle\vec{x}\rangle : q|] &\overset{\text{def}}{=} x.l\langle\vec{x}\rangle, [|q|]\\
[|x = Me : q|] &\overset{\text{def}}{=} x = eM, [|q|].
\end{aligned}$$

For example, the environment state

$$\mathbf{run}\ [x = \{l = (y)[]\}]\{x/x\}\ \mathbf{in}\ x = l\langle 1\rangle : l\langle 2\rangle \qquad (3)$$

is translated into the simple state

$$\mathbf{run}\ [x = \{l = (y)[]\}]\ \mathbf{in}\ x.l\langle 1\rangle, x.l\langle 2\rangle. \qquad (4)$$

We are now in a position to present the main result of this section.

Theorem 3 (Soundness).

1. *If* $E \rightarrow E'$, *then* $[|E|] \rightarrow E'$;
2. *If* $\mathbf{run}\ T_0\emptyset\ \mathbf{in}\ \varepsilon \downarrow E$, *then* $[|E|] \not\rightarrow$.

Proof.

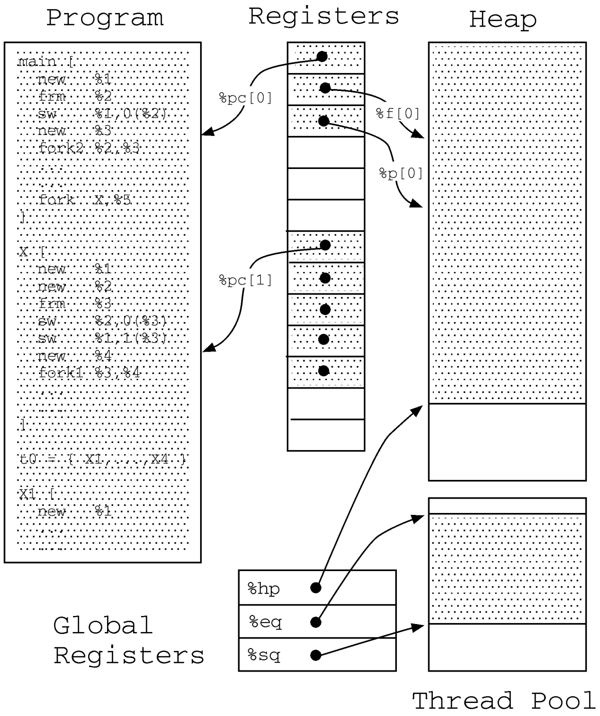
1. By induction on the structure of a reduction step. Rules in the simple and in the environment machine are in a one-to-one correspondence.
2. Similar to the proof of the second clause of Theorem 1, this time using the invariant discussed in Section 5.4. □

The environment machine is not complete with respect to the simple machine (hence, with respect to TyCO) since we now impose a discipline in the access to the store. In fact, there are reductions of the simple machine that cannot be mimicked by the environment machine. For example, the simple state $[|E|]$ described in (4) above may reduce to state $\mathbf{run}\ []\ \mathbf{in}\ x.l\langle 1\rangle$, whereas E (3) can only reduce to state $\mathbf{run}\ []\ \mathbf{in}\ x.l\langle 2\rangle$.

7 Implementation

We propose an implementation that closely follows the specification of the abstract machine in the previous section.

1. Each thread T is compiled into a block of contiguous instructions. The machine starts with the initial thread (the program) and an empty store, that is, with the initial state $\mathbf{run}\ T_0\emptyset\ \mathbf{in}\ \epsilon$.
2. The store s is implemented with a heap datastructure. The environment e of a thread T is implemented as a table in the heap and is copied into registers when T is running.
3. Resource closures, Me and Ae, are implemented as pairs residing in the heap and composed of a reference to a piece of code, the dispatch table for M or the code for A, and a table holding the environment e.
4. The code for an abstraction A is just a block of contiguous instructions; that of a method map M is composed of a dispatch table followed by the code for each method.
5. Tokens $X\langle\vec{x}\rangle$, $y.l\langle\vec{x}\rangle$ are implemented as tables in the heap holding the arguments $\vec{x}$ (plus the label l in the case of messages).
6. Thread closures $Te'e''$ in the pool have environments divided in two tables, arguments e' and free variables e'' coming, respectively, from a token and from a resource (see rules FORK).
7. A new thread closure, c, is added to the thread pool when a fork occurs. New threads can be started when the current one ends (rule GC) or when a context switch occurs (rule SWITCH).
8. The machine halts when the thread pool is empty, that is, when the final state $\mathbf{run}\ \epsilon\ \mathbf{in}\ s$ is reached.



```
main [
   new   %1
   frm   %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
   ...
   ...
```

Fig. 5. Machine architecture.

7.1 Machine Architecture

We propose a heap based machine architecture (Fig. 5). Each thread keeps a small amount of state in its environment, namely, the program counter and the location of the arguments and free variables in the heap. Global registers keep track of the top of the heap and of the limits of the thread pool. Each instance of the program counter register points into the program area where the code for the complete program is stored. A set of generic registers, denoted `%i`, `%j`, where `i`, `j` are nonnegative integers, is used to keep the instruction operands. We assume the number of registers to be as large as needed. Hardware or software techniques such as register renaming or register windows may be used to provide this illusion in the real world. Data is moved between the heap and machine registers by common load/store instructions. Thus, in a sense, the instruction set architecture defined below is RISC-like.

The *program area* keeps the code for the program, the byte-code for thread T_0, to be executed. The code is divided in thread blocks and dispatch tables for objects. The *heap* is a flat address space where dynamic data-structures are allocated. Space is allocated in blocks of contiguous machine words called frames. The machine manipulates three basic data-structures at runtime: tags (x), tokens ($X\langle\vec{y}\rangle$, $l\langle\vec{y}\rangle$), and resources (M, A). Tags index shared queues of message tokens $l\langle\vec{x}\rangle$ and of method closures Me. Message tokens are implemented as frames holding the label l plus a variable number of arguments $\vec{x}$. Method closures Me are implemented as frames holding a reference to a dispatch table plus an environment table for the free variables in M.

The *thread pool* implements a collection of thread closures $\tilde{c}$ ready for execution. It is used to account for the limited resources and performance considerations present in real machines, imposing an upper bound on the number of simultaneously active threads. Each thread closure, $Te'e''$, holds a reference to the piece of code for T in the program and references to the parameter (e') and free variable (e'') environments in the heap. A new thread, resulting from a FORK, inherits its environment from the token and the resource and is placed in the pool waiting to be scheduled for execution.

Environment variables, x, introduced with NEW are initially bound to empty queues in the heap and allocated to generic machine registers. They are discarded after the thread terminates. Despite their short life span, the tags they are bound to may continue to exist long after the thread ends. This is accomplished, for example, when a tag is sent as an argument to a message targeted outside the scope of the tag, thus escaping the context of the current thread.

7.2 Special Registers

The machine uses a small set of global registers to control the main data-structures, namely the heap and the thread pool. Register `%hp` (Heap Pointer) points to the next available position in the heap. Registers `%eq` and `%sq` keep the boundaries of the thread pool. The environment of each thread is kept in three special local registers. Register `%pc` (Program Counter) points to the next instruction to be executed in the thread. When a program starts, register `%pc` is loaded with the address of the first instruction of the `main` thread. Registers `%f` (free variable environment) and `%p` (parameter environment) are used to hold references to the free variable and parameter environments, respectively.

7.3 Instruction Set Architecture

The core instruction set is described below.

`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

Heap allocation instructions allocate space for data-structures. `frm %i,n` allocates a frame of size `n` in the heap and keeps a pointer to it in register `%i`. `new %i` creates a new queue in the heap and keeps a pointer to it in the register `%i`.

Load/Store instructions move data from the heap into registers and vice-versa. `lw %i,k(%j)` copies the word at the heap frame pointed to by `%j` and at offset `k` to the register `%i`. `sw %i,k(%j)` copies the word at the register `%i` to the heap frame pointed to by `%j`, at offset `k`.

Fork instructions implement reduction. `forkd X,%i` creates a new thread running the code labeled by `X` and parameters pointed to by register `%i`. `forko %i,%j` takes an object at a frame pointed to by register `%i` and tries to reduce at the tag pointed to by register `%j`. `forkm %i,%j` takes a message at a frame pointed to by register `%i` and tries to reduce at the tag pointed to by register `%j`.

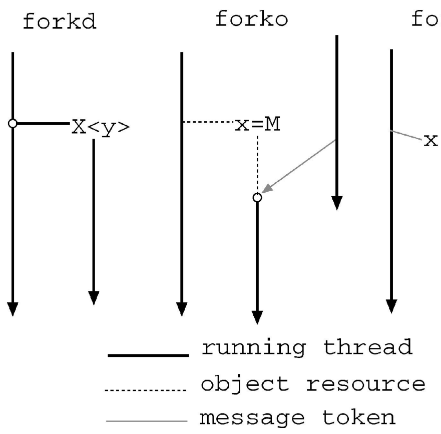


Fig. 6. The three fork operations.

Multithreading instructions manage the execution of the multiple threads generated by the machine at runtime. `switch n` performs a context switch to the `n`th thread in the pool and places the current thread back in the pool. `newt` terminates the execution of the current thread and loads a new one from the pool.

The three `fork` instructions are illustrated in Fig. 6. A `forkd` immediately creates a new thread that is added to the thread pool. A `forko` instruction generates a method closure that requires an extra message token to produce a runnable thread. When (if) such a message arrives, a new thread is activated and is added to the pool. Finally, `forkm` instructions generate message tokens that remain in the heap waiting for a suitable object resource. When this happens, a new thread is added to the pool.

These instructions are then complemented with the usual set of control flow instructions, including relative and absolute jump instructions, to control the flow within a thread and primitive instructions for arithmetic and logic operations. All these instructions are register-to-register, as is usual in RISC designs.

8 Multithreaded Code

This section sketches the compilation of high level programming languages into the TTyCO calculus and, finally, into the assembly of the abstract machine. We claim that TTyCO can be used as an effective intermediate representation language for many high level idioms and that this translation allows implicit parallelism to be exposed in the form of multithreaded programs.

Several idioms have been encoded into TTyCO, namely a functional core [28], an idiom for client-server/session based computing [12], and a language with support for distribution and code mobility [30]. In the sequel, we will use an example from a functional language to describe the compilation scheme for TTyCO. The example consists of the well-known map function applied to a tree structure.

```
datatype Tree = { bud, leaf Int, node Tree Tree }

fun map f t = case t of {
    bud = bud,
    leaf n = leaf (f n),
    node l r = node (map f l) (map f r)
}
```

The above definition can be encoded into a TTyCO program by a straightforward encoding [28]. We get the following intermediate representation:

```
$t_0$[
    Map = (f,t,map-reply) $t_1$[
        new case-reply; t.val⟨case-reply⟩;
        case-reply = {
            bud = () $t_2$[
                new x; Bud⟨x⟩; map-reply.val⟨x⟩
            ],
            leaf = (n) $t_3$[
                new x; f.val⟨n,x⟩;
                x = {val = (v) $t_5$[
                    new y; Leaf⟨y,v⟩; map-reply.val⟨y⟩
                ] }
            ],
            node = (l,r) $t_4$[
                new x; Map⟨f,l,x⟩;
                new y; Map⟨f,r,y⟩;
                x = {val = (lt) $t_6$[
                    y = {val = (rt) $t_7$[
                        new z;Node⟨z,lt,rt⟩;map-reply.val⟨z⟩
                    ] }
                ] }
            ]
        }
    ]
    ... // use map
]
```

The Assembly Layout closely follows that of the source programs in the sense that threads constitute the main organization block for the code. The code for nested threads in an assembly program is flattened. A typical code block for a thread $[I_1; \ldots ; I_n]$ is shown below:

```
thread label [
    load arguments
    load free variables
    code for I$_1$
    ...
    code for I$_n$
    get new thread
]
```

Each thread is identified by a unique `label`. Notice that the only occasion where `lw` instructions are used is at the beginning of the execution of a thread, except for eventual *spilling* events. The code for each of the instructions I_i is composed of a sequence of basic machine instructions.

Compiling objects requires the use of dispatch tables. They appear in between thread blocks in the assembly code and are identified by unique `labels`. Each label in such a sequence holds a pointer to the code of some method in an object.

$\texttt{label} = \{\texttt{l}_1, \ldots, \texttt{l}_n\}$

The Compiler is fairly typical in that it recursively flattens the nested threads in the TTyCO encoding of the high level construct into a sequence of independent, single code block, threads.

The breakup into threads is very noticeable in the intermediate TTyCO code and is even more apparent when we proceed one further step and compile it to our target instruction set architecture. Here is the code for the central object controlling the **case** statement:

```
o1 = { t2 , t3, t4 }      // the dispatch table

thread t2 [               // the bud case
  lw    %0,1(%f)          // map-reply
  new   %1                // new x
  frm   %2,1
  sw    %1,0(%2)
  forkd Bud,%2            // Bud<x>
  frm   %3,2
  sw    0,0(%3)
  sw    %1,1(%3)
  forkm %3,%0             // map-reply.val<x>
  newt
]

thread t3  [              // the leaf case
  lw      %0,0(%p)        // n
  lw      %1,0(%f)        // f
  lw      %2,1(%f)        // map-reply
  new     %3              // new x
  frm     %4,3
  sw      0,0(%4)
  sw      %0,1(%4)
  sw      %3,2(%4)
  forkm   %4,%1           // f.val<n,x>
  frm     %5,2
  sw      t5,0(%5)
  sw      %2,1(%5)
  forko   %5,%3           // x={val= (v)t5[...]
  newt
]

thread t4  [              // the node case
  lw      %0,0(%p)        // l
  lw      %1,1(%p)        // r
  lw      %2,0(%f)        // f
  lw      %3,1(%f)        // map-reply
  new     %4              // new x
  frm     %5,3
  sw      %2,0(%5)
  sw      %0,1(%5)
  sw      %4,2(%5)
  forkd  Map,%5           // Map<f,l,x>
  new     %6              // new y
  frm     %7,3
  sw      %2,0(%7)
  sw      %1,1(%7)
  sw      %6,2(%7)
  forkd  Map,%7           // Map<f,r,y>
  frm     %8,3
  sw      t6,0(%8)
  sw      %3,1(%8)
  sw      %6,2(%8)
  forko   %8,%4           // x={val= (lt)t6[...]
  newt
]
```

9 Conclusions and Further Work

Multithreading is an important technique that is very likely to migrate to hardware in the next generations of microprocessors, opening new possibilities in the exploitation of fine-grained parallelism in applications. From a programming point of view, process-calculi provide a suitable paradigm not only to formally model such systems but also to provide compilation schemes that naturally break down high level programs into ISA level threaded code particularly suitable for these hardware architectures.

The programming languages more akin to TTyCO also derive from the realm of the process calculi. Pict [24] is a pure concurrent programming language based on the asynchronous π-calculus. The runtime system is based on Turner's abstract machine specification [27]. The basic programming abstractions are processes and names (tags). Processes communicate by sending values along shared names. Objects in Pict are less efficient than in TTyCO. They require more heap space and have a more complex method invocation protocol, involving two messages. Turner's machine also uses replication for persistent data, producing substantially more heap garbage than TTyCO, which uses recursion.

Another related language is Join, an implementation of the Join calculus [9]. Names (tags), expressions, and processes are the basic abstractions. Join programs are composed of processes, communicating asynchronously on names and producing no values, and expressions evaluated synchronously and producing values. Join introduces a powerful extension—the *join-pattern*. Patterns combine names, input processes, and replication into a single construct. A join-pattern defines a synchronization pattern between input processes waiting on a collection of names.

On the more conventional side, Cilk is the project most akin to ours. Cilk [5] is an efficient multithreaded runtime system developed at MIT. Cilk computations may be viewed as directed acyclic graphs that evolve in time. Programs are composed of a sequence of *procedures*, each of which is broken into a sequence of one or more *threads*. Threads are nonblocking, which means a thread cannot spawn children and wait for their results. The computation evolves in a data-flow fashion. The Cilk language is an extension to C that provides an abstraction of threads in explicit continuation-passing style. Cilk programs are preprocessed to C code and then linked with a runtime library.

Currently, we have a sequential implementation of a fine-grained, object-based language based in the TTyCO calculus. This kernel language features objects, asynchronous method invocations, and threads as main abstractions. The language is strongly, implicitly typed and supports parametric polymorphism. The abstract machine is implemented in the form of a compact, portable, and self-contained byte-code emulator. The semantics is provided by the formal model presented in Section 5, which itself grows from Turner's work on the π-calculus. The main novelty is the introduction of explicit threads as computational units. This makes compiler support for very fine-grained multithreading possible as it exposes parallelism at the ISA level.

We have shown, through a set of experiments, that our implementation is quite efficient even by comparison with systems that compile directly to C or native code [15], [16]. TTyCO performs close to Pict [24] and Oz [26] in programs that are mainly functional, whereas, in programs with nontrivial object data structures, it outperforms them both in speed and heap usage.

The work on the TTyCO language focuses on three areas. First, we aim at implementing a fully multithreaded system starting from the current sequential implementation. Using this model, we wish to study the opportunities for fine-grained parallelism, namely in the presence of simple interleaving and true thread parallelism. Finally, an interesting point concerns the implications of multithreaded execution in the development of type systems that could allow interesting type driven optimizations.

Acknowledgments

The authors wish to thank the anonymous referees for their valuable comments. This work was partially supported by project DiCoMo (contract PRAXIS/P/EEI/12059/98) and by funds granted to LIACC through the Programa de Financiamento Plurianual, FCT, and Programa POSI. This is an extended version of a paper appearing in the *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, pp. 217-226.

References

[1] A. Agarwal et al., "SPARCLE: A Multithreaded VLSI Processor for Parallel Processing," *Lecture Notes in Computer Science,* vol. 748, p. 395, Springer-Verlag, 1993.

[2] R. Alverson et al., "The TERA Computer System," *Proc. Int'l Conf. Supercomputing—ICS '90,* pp. 1-6, 1990.

[3] Arvind and V. Kathail, "A Multiple Processor Dataflow Machine that Supports Generalized Procedures," *Proc. Eighth Int'l Symp. Computer Architecture,* pp. 291-302, 1981.

[4] H.G. Baker, "'Use-Once' Variables and Linear Objects-Storage Management, Reflection and Multi-Threading," *ACM SIGPLAN Notices,* vol. 30, no. 1, p. 45, 1995.

[5] R.D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," *ACM SIGPLAN Notices,* vol. 30, no. 8, pp. 207-216, 1995.

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

[7] G. Boudol, "The π-Calculus in Direct Style," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '97),* 1997.

[8] D. Culler et al., "TAM—A Compiler Controlled Threaded Abstract Machine," *J. Parallel and Distributed Computing,* vol. 18, no. 3, pp. 347-370, 1993.

[9] C. Fournet and G. Gonthier, "The Reflexive Chemical Abstract Machine and the Join-Calculus," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '96),* pp. 372-385, 1996.

[10] J. Gurd and I. Watson, "A Multi-Layered Dataflow Computer Architecture," *Proc. Int'l Conf. Parallel Programming—ICPP '77,* p. 94, 1977.

[11] K. Honda and M. Tokoro, "An Object Calculus for Asynchronous Communication," *Proc. European Conf. Object-Oriented Programming (ECOOP '91),* pp. 141-162, 1991.

[12] K. Honda, V. Vasconcelos, and M. Kubo, "Language Primitives and Type Disciplines for Structured Communication-Based Programming," *Proc. European Symp. Programming (ESOP '98),* pp. 122-138, 1998.

[13] N. Kobayashi, "Quasi-Linear Types," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '99),* pp. 29-42, 1999.

[14] N. Kobayashi, B. Pierce, and D. Turner, "Linearity and the π-Calculus," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '96),* 1996.

[15] L. Lopes, "On the Design and Implementation of a Virtual Machine for Process Calculi," PhD thesis, Faculty of Sciences, Univ. of Porto, Portugal, 1999. Available from: http://www.ncc.up.pt/~lblopes/.

[16] L. Lopes, F. Silva, and V. Vasconcelos, "A Virtual Machine for the TyCO Process Calculus," *Proc. Principles and Practice of Declarative Programming (PPDP '99),* pp. 244-260, 1999.

[17] J. McGraw et al., *The SISAL Language Reference Manual—Version 1.2,* 1985.

[18] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[19] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes (Parts I and II)," *Information and Computation,* vol. 100, no. 1, pp. 1-77, 1992.

[20] R. Nikhil, "The Parallel Programming Language Id and Its Compilation for Parallel Machines," *Int'l J. High Speed Computing,* vol. 5, pp. 171-223, 1993.

[21] R. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing," *Proc. 16th Int'l Symp. Computer Architecture,* pp. 262-272, 1989.

[22] R. Nikhil, G. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Int'l Symp. Computer Architecture,* pp. 156-167, 1992.

[23] G. Papadopoulos and D. Culler, "Monsoon: An Explicit Token-Store Architecture," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 82-91, 1990.

[24] B. Pierce and D. Turner, "Pict: A Programming Language Based on the Pi-Calculus," Technical Report CSCI 476, Computer Science Dept., Indiana Univ., 1997.

[25] B. Smith, "A Pipelined, Shared Resource MIMD Computer," *Proc. Int'l Conf. Parallel Programming—ICPP '78,* pp. 6-8, 1978.

[26] G. Smolka, "The Oz Programming Model," *Computer Science Today,* pp. 324-343, Springer-Verlag, 1995.

[27] D. Turner, "The Polymorphic Pi-Calculus: Theory and Implementation," PhD thesis, Univ. of Edinburgh, 1995.

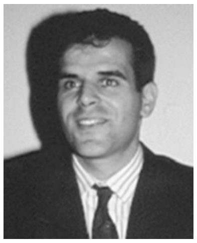
[28] V. Vasconcelos, "Processes, Functions, Datatypes," *Theory and Practice of Object Systems,* vol. 5, no. 2, pp. 97-110, 1999.

[29] V. Vasconcelos and R. Bastos, "Core-TyCO—The Language Definition," Technical Report TR-98-3, Dept. of Informatics, Univ. of Lisbon, 1998.

[30] V. Vasconcelos, L. Lopes, and F. Silva, "Distribution and Mobility with Lexical Scoping in Process Calculi," *Proc. Workshop High Level Programming Languages (HLCL '98),* pp. 19-34, 1998.

[31] V. Vasconcelos and M. Tokoro, "A Typing System for a Calculus of Objects," *Proc. Int'l Symp. Object Technologies for Advanced Software (ISOTAS '93),* pp. 460-474, 1993.

Luís Lopes received the BSc degree in applied mathematics and computer science from the University of Porto, Portugal, in 1992 and the PhD degree in computer science from the same university in 1999. He is an assistant professor in the Department of Computer Science at the University of Porto. His research interests include distributed and mobile computing, parallel computing, programming languages, and computer architecture.



Vasco T. Vasconcelos received the BSc degree in informatics from the New University of Lisbon, Portugal, in 1988. He received the MSc and PhD degrees in computer science from the University of Keio, Japan, in 1992 and 1994, respectively. He is an assistant professor in the Department of Informatics, University of Lisbon. His research interests include programming languages, concurrency, type systems, and distributed systems.

Fernando Silva received the BSc degree in applied mathematics from the University of Porto, Portugal, in 1985. He received the MSc and PhD degrees in computer science from the University of Manchester, United Kingdom, in 1988 and 1993, respectively. He is an assistant professor in the Department of Computer Science at the University of Porto. His research interests include parallel and distributed computing, parallel logic programming, mobile computing, and programming languages. He is currently the director for the Southwestern Europe ACM Programming Contest.

▷ **For further information on this or any computing topic, please visit our Digitial Library at** http://computer.org/publications/dlib.