Fine Grained Multithreading with Process Calculi

Luís Lopes, Fernando Silva
Dep. of Computer Science & LIACC
Faculty of Sciences, University of Porto
e-mail: {lblopes,fds}@ncc.up.pt

Vasco T. Vasconcelos
Department of Informatics
Faculty of Sciences, University of Lisbon
e-mail: vv@di.fc.ul.pt

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 the construction of formally verifiable systems on which to base high-level programming idioms, combined with efficient compilation schemes into multithreaded architectures.

Keywords: Process-Calculus, Multithreading, Abstract-Machine.

1. Introduction

In recent years researchers have devoted a great effort in providing semantics for pure concurrent/parallel programming languages within the realm of process-calculi. Milner, Parrow and Walker's π-calculus or a asynchronous formulation due to Honda and Tokoro [11, 20] and Boudol [6] has 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, 32].

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, 24], hybrid dataflow/control-flow [8, 22, 23] and lately multithreaded RISC [1, 2, 26] designs. Multithreading aims to provide high processor utilization in the presence of large memory or interprocessor communication latency. These 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, multithreading hardware support 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 hardware multithreading would then overlap the multiple threads from that single application.

Programming paradigms and compilations techniques adequate 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 [21], to numerically oriented Sisal [18] 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 [22].

Languages that allow an efficient compilation 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, 32] 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 when a high latency load or interprocessor communication occurs). Finally, these calculi also commonly feature type systems that allow not only to check the type safeness of programs but also to collect 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 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. Section 5 describes an implementation for the second abstract machine, namely the fundamental data structures and the instruction set architecture. Section 6 describes a compilation scheme for languages using the calculus as an intermediate representation language. Finally, section 7 compares this approach to other related work and issues some conclusions and future research directions.

2. The Threaded TyCO Calculus

We start with the TyCO (Typed Concurrent Objects) process calculus [30, 32]. TyCO is a form of the asynchronous π-calculus featuring first class objects, asynchronous messages, and polymorphic process definitions. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. Here, we recast the calculus 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 $\alpha_1, \ldots, \alpha_n$, and let $\vec{\alpha}$ denote $\alpha_1; \ldots; \alpha_n$, for $n \geq 0$. When n is 0, $\tilde{\alpha}$ is represented by ε (no special notation is needed in this case for $\vec{\alpha}$). The difference between $\tilde{\alpha}$ and $\vec{\alpha}$ is that we allow permutations on the former, but not on the latter.

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, X is the tag of the resource. Tokens are pieces of data that activate resources, ultimately producing new threads. A token is an asynchronous message $x.l\langle\tilde{y}\rangle$ that invokes the method l in an object resource tagged with x, or 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 variables. Think of these variables 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 disjoint sets for object tags (x, y, z, infinite), for thread tags (X), and for labels (l), we write the full syntax of the calculus as follows.

$$\begin{array}{rcll}
T & ::= & [\vec{I}] & \text{Thread} \\
I & ::= & t \mid R \mid \textbf{new}\, x & \text{Instruction} \\
t & ::= & x.l\langle\vec{x}\rangle \mid X\langle\vec{x}\rangle & \text{Token} \\
R & ::= & x = M \mid X = A & \text{Resource} \\
M & ::= & \widetilde{l = A} & \text{Method map} \\
A & ::= & (\vec{x})T & \text{Abstraction}
\end{array}$$

To illustrate the syntax and main abstractions of the calculus we sketch a small example of a single element polymorphic cell. We define a thread abstraction, the Cell, with two attributes: the location self and the value v 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.

```
[
    Cell = (self, value) [
        self = {
            read = (reply-to) [
                reply-to.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⟩;
    r = {val = (x)[io.print⟨x⟩] }
]
```

TTyCO features a predicative polymorphic type-system and, as a result, the Cell abstraction is polymorphic on the attribute value. Next, we create instances of Cell with integer and boolean attributes, respectively. 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 An Abstract Machine

Here we present an abstract machine for interpreting TTyCO processes. 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. Before we present the reduction relation, some machinery must be introduced.

Preliminaries. We take the view that programs are closed for tags, that is that the collection of threads that constitutes a program contains no free tags. A tag x is bound in the sequence of instructions $\vec{I}'$ in a thread of the form $[\vec{I};$ **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. A tag X is bound in A and in $\vec{I}'$ in a thread of the form $[\vec{I}; X = A; \vec{I}']$; otherwise it is free.

A direct consequence of this definition is that, in a closed program, when scanning 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 will never find their way into the store.

Another consequence is that we cannot have mutually recursive thread definitions. A small change in the syntax of resources, replacing $X = A$ by $\widetilde{X = A}$ (maps from thread tags into abstractions), would solve the problem. For simplicity we proceed with simple recursive thread definitions $X = A$.

Items in the store are a subset of the possible instructions. We make this clear by introducing a new syntactic category.

$$S \quad ::= \quad R \;\mid\; x.l\langle\vec{y}\rangle$$

Then, the state of the machine is represented by a term **run** $\tilde{T}$ **in** $\tilde{S}$ denoting a pool of threads running on a pool of available resources and tokens. We assume that all bound tags (object and thread) in the initial program are pairwise distinct. Given such a program $\tilde{T}_0$ we build the initial state of the machine as **run** $\tilde{T}_0$ **in** ε.

Given the above notion of free tags, we denote by $\{\vec{x}/\vec{y}\}T$ the usual (capture-avoiding) substitution of $\vec{x}$ for $\vec{y}$ in T. To extract a method in a map of methods, we use $(l = A, M).l \overset{\text{def}}{=} A$; to apply an abstraction to a sequence we write $((\vec{y})T)\langle\vec{x}\rangle \overset{\text{def}}{=} \{\vec{x}/\vec{y}\}T$; so that, when M is $(l = (\vec{y})T, M')$, the expression $M.l\langle\vec{x}\rangle$ stands for the thread $\{\vec{x}/\vec{y}\}T$.

Structural congruence. Following Milner [19] we divide the computational rules of the calculus in two parts: the *structural congruence* rules and the *reduction* rules. Structural congruence rules allow the re-writing 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 threads that includes the following single rule.

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

Notice that concurrency and non-determinism are introduced by allowing arbitrary permutations between threads and items in the store.

Reduction. 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 of tags in the store. Figure 1 summarizes the rules.

FORK-M: A method invocation $x.l\langle\vec{y}\rangle$ 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 thread whose parameters have been replaced by the arguments $\vec{y}$. FORK-O: Inversely, an object $x = M$ is triggered by a message

$$\textsf{\textbf{run}}\ [x.l\langle\vec{y}\rangle;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ x = M,\tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T},M.l\langle\vec{y}\rangle\ \textsf{\textbf{in}}\ \tilde{S} \qquad (\textsc{fork-m})$$

$$\textsf{\textbf{run}}\ [x = M;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ x.l\langle\vec{y}\rangle,\tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T},M.l\langle\vec{y}\rangle\ \textsf{\textbf{in}}\ \tilde{S} \qquad (\textsc{fork-o})$$

$$\textsf{\textbf{run}}\ [X\langle\tilde{y}\rangle;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ X = A,\tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T},A\langle\vec{y}\rangle\ \textsf{\textbf{in}}\ X = A,\tilde{S} \qquad (\textsc{fork-d})$$

$$\textsf{\textbf{run}}\ [x = M;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ x = M,\tilde{S} \qquad (\textsc{store-o})$$

$$\textsf{\textbf{run}}\ [x.l\langle\vec{y}\rangle,\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ x.l\langle\vec{y}\rangle,\tilde{S} \qquad (\textsc{store-m})$$

$$\textsf{\textbf{run}}\ [X = A;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ [\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ X = A,\tilde{S} \qquad (\textsc{store-d})$$

$$\textsf{\textbf{run}}\ [\textsf{\textbf{new}}\ x;\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ [\{y/x\}\vec{I}],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \quad \text{if } y \text{ not free in } \vec{I},\tilde{T},\tilde{S} \qquad (\textsc{new})$$

$$\textsf{\textbf{run}}\ [\,],\tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ \tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \qquad (\textsc{gc})$$

$$\frac{\tilde{S} \equiv \tilde{S}' \qquad \tilde{T} \equiv \tilde{T}' \qquad \textsf{\textbf{run}}\ \tilde{T}'\ \textsf{\textbf{in}}\ \tilde{S}' \to \textsf{\textbf{run}}\ \tilde{T}''\ \textsf{\textbf{in}}\ \tilde{S}''}{\textsf{\textbf{run}}\ \tilde{T}\ \textsf{\textbf{in}}\ \tilde{S} \to \textsf{\textbf{run}}\ \tilde{T}''\ \textsf{\textbf{in}}\ \tilde{S}''} \qquad (\textsc{struct})$$

Figure 1. Abstract machine for TTyCO

$x.l\langle\vec{y}\rangle$ 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 we create a new instance $X\langle\vec{x}\rangle$ of a thread definition $X = A$. 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 wouldn't 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.

NEW: To create a new tag, we rely on the set of tags being infinite, and pick one unused in the rest of the state. GC allows the garbage collection of an empty thread. STRUCT brings structural congruence into reduction.

The machine halts when the pool of threads empties, that is, when a state **run** ε **in** $\tilde{S}$ is reached.

Discussion. With respect to TyCO, we have traded the explicit parallel composition of processes for the sequential nature of threads, coupled with the non-determinism in the selection of a resource or a token (captured by the structural congruence relation). We expect threaded TyCO to be sound and complete with respect to TyCO.

Also, for typable programs [32] the abstract machine does not deadlock, that is, it either halts or runs indefinitely [15]. This property is intimately linked to the ability of the type system to guarantee that, in a typable program, objects have the right methods for messages [17].

4 A Refinement of the Abstract Machine

The machine proposed in the previous section is still far from an efficient interpreter: substitutions are actually performed by visiting the threads, and the store is completely unstructured, the machine relying 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 for directly accessing the resources and tokens in the store. Once again, some new definitions are needed before we may present the reduction relation.

Preliminaries. 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 to (thread) abstraction closures Ae, and from object tags x to queues q of method closures or of messages contents $l\langle\vec{x}\rangle$. A queue q can be regarded as a possibly empty list $\alpha_1 : \cdots : \alpha_n$; 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 machine is now represented by a term **run** $\tilde{c}$ **in** s denoting a pool of thread closures running on a map from tags to the available resources and tokens.

$$\textsf{\textbf{run}}\ [x.l\langle\vec{y}\rangle;\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c},Te'\{\vec{z}:=e(\vec{y})\}\ \textsf{\textbf{in}}\ s\{e(x):=q\} \quad \text{if } s(e(x))=Me':q, M.l=(\vec{z})T \qquad (\text{FORK-M})$$

$$\textsf{\textbf{run}}\ [x=M;\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c},Te\{\vec{z}:=\vec{y}\}\ \textsf{\textbf{in}}\ s\{e(x):=q\} \quad \text{if } s(e(x))=l\langle\vec{y}\rangle:q, M.l=(\vec{z})T \qquad (\text{FORK-O})$$

$$\textsf{\textbf{run}}\ [X\langle\vec{y}\rangle;\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c},Te'\{\vec{z}:=e(\vec{y})\}\ \textsf{\textbf{in}}\ s \quad \text{if } s(X)=((\vec{z})T)e' \qquad (\text{FORK-D})$$

$$\textsf{\textbf{run}}\ [x=M;\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s\{e(x):=q:Me\} \quad \text{if } s(e(x))=q \qquad (\text{STORE-O})$$

$$\textsf{\textbf{run}}\ [x.l\langle\vec{y}\rangle,\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s\{e(x):=q:l\langle e(\vec{y})\rangle\} \quad \text{if } s(e(x))=q \qquad (\text{STORE-M})$$

$$\textsf{\textbf{run}}\ [X=A;\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ [\vec{I}]e,\tilde{c}\ \textsf{\textbf{in}}\ s\{X:=Ae\} \qquad (\text{STORE-D})$$

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

$$\textsf{\textbf{run}}\ \tilde{c}_1,c,\tilde{c}_2\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ c,\tilde{c}_1,\tilde{c}_2\ \textsf{\textbf{in}}\ s \qquad (\text{SWITCH})$$

$$\textsf{\textbf{run}}\ []e,\tilde{c}\ \textsf{\textbf{in}}\ s \to \textsf{\textbf{run}}\ \tilde{c}\ \textsf{\textbf{in}}\ s \qquad (\text{GC})$$

Figure 2. The environment machine for TTyCO

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.

Reduction. Figure 2 summarizes the 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 $e(x)$. A new thread closure is created with the appropriate environment; Me' is dequeued. 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, only 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 previous machine, the difference being that tokens and resources are now stored at the tail of the queue associated with the tag. This time, 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.

SWITCH: This new rule reintroduces non-determinism via permutations of thread closures (notice that rule STRUCT is no longer present). It states that at any point in the execution we may switch the context to run another thread c anywhere in the pool. The current thread is accordingly moved back to the thread pool. This rule is fundamental for the implementation of multithreading.

Discussion. Because we have imposed a discipline (FIFO) in the access to the resources in the store as well as an order in the application of the FORK/STORE rules, we expect the environment machine not to be complete with respect to the machine in the previous section. The following run

$$\textsf{\textbf{run new}}\ x; x=\{l=(y)[]\}; x.l\langle 1\rangle; x.l\langle 2\rangle;\ \textsf{\textbf{in}}\ \varepsilon \to^5 \\ \textsf{\textbf{run}}\ \varepsilon\ \textsf{\textbf{in}}\ x.l\langle 1\rangle$$

may not be mimicked by the environment machine.

Also, the machine preserves the invariant that, at any time during a computation the queues are either empty or have only messages or have only method-closures [25].

5. Implementation

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

1. Each thread is compiled into a block of contiguous instructions. The machine starts with the initial thread and an empty store.

2. The store is implemented with multiple data-structures residing in a heap. Environments are implemented as tables in the heap, and are copied to registers when the thread is running.

3. Resource closures are implemented as pairs residing in the heap, and composed of a reference to a piece of code and a table holding the environment.

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).

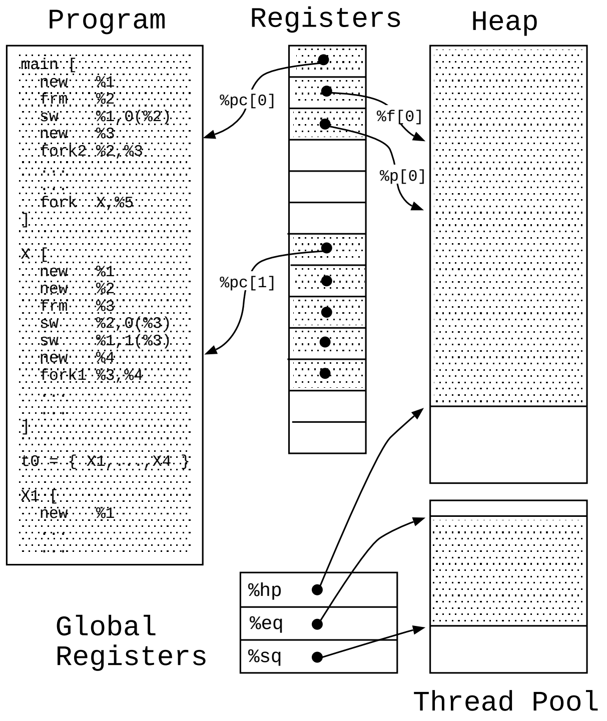
6. Thread closures in the pool have environments divided in two tables, arguments and free variables, coming from a

token and from a resource, respectively.

7. A new thread closure, its environment built as described above, 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.

Machine Architecture We propose a heap based machine architecture (Figure 3). 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 non negative 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.



```
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
  ...
  ...
```

Figure 3. Machine architecture

The *program area* keeps the code for the program 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: (object) tags, tokens and resources. 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 holds a reference to a piece of code in the program and references to the parameter and free variable environment 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 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.

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.

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`.

Multithreading instructions manage the execution of the multiple threads generated by the machine at run-time. `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 figure 4. 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.

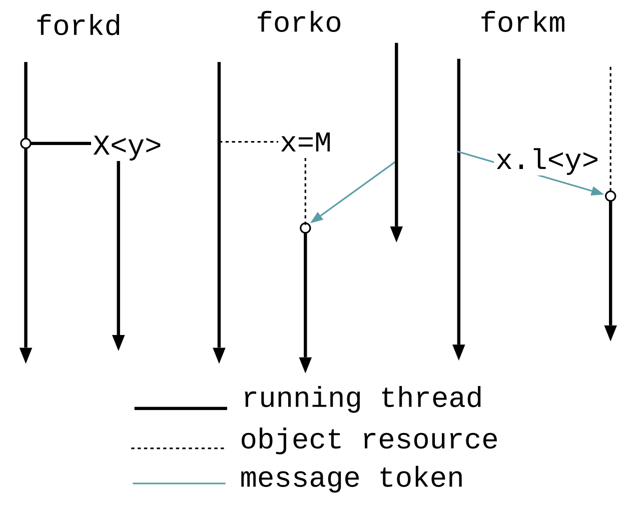


Figure 4. The three fork operations

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.

6. Multithreaded Code

This section describes 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 [29], an idiom for client-server/session based computing [12] and, a language with support for distribution and code mobility [31]. 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.

```
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 [29]. We get the following intermediate representation:

```
t0[
    Map = (f,t,map-reply) t1[
        new case-reply; t.val⟨case-reply⟩;
        case-reply = {
            bud = () t2[
                new x; Bud⟨x⟩; map-reply.val⟨x⟩
            ],
            leaf = (n) t3[
                new x; f.val⟨n,x⟩;
                x = {val = (v) t5[
                    new y; Leaf⟨y,v⟩; map-reply.val⟨y⟩
                ] }
            ],
            node = (l,r) t4[
                new x; Map⟨f,l,x⟩;
                new y; Map⟨f,r,y⟩;
                x = {val = (lt) t6[
                    y = {val = (rt) t7[
                        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 I₁
  ...
  code for Iₙ
  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 `label`s. Each label in such a sequence holds a pointer to the code of some method in an object.

```
label = {l₁,...,lₙ}
```

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
]
```

7 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 [25] is a pure concurrent programming language based on the asynchronous π-calculus. The run-time system is based on Turner's abstract machine specification [28]. 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 that 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 a 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. Cilk programs are composed of a sequence of *procedures* each of which is broken into a sequence of one or more *threads*. Threads are non-blocking, 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 run-time 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 4, 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 [25] and Oz [27] in programs that are mainly functional whereas in programs with non-trivial 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.

Acknowledgements. The authors wish to thank the anonymous referees for their valuable comments.

References

[1] A. Agarwal, J. Babb, and et.al. SPARCLE: A Multithreaded VLSI Processor for Parallel Computing. Number 748 in LNCS, page 395. Springer-Verlag, 1993.

[2] R. Alverson, D. Callahan, and et.al. The TERA Computer System. In *International Conference on Supercomputing - ICS'90*, pages 1–6, 1990.

[3] Arvind and V. Kathail. A Multiple Processor Dataflow Machine that Supports Generalized Procedures. In *8th International Symposium on Computer Architecture*, pages 291–302, 1981.

[4] H. G. Baker. 'Use-Once' Variables and Linear Objects-Storage Management, Reflection and Multi-Threading. *ACM SIGPLAN Notices*, 30(1):45, 1995.

[5] R. D. Blumofe, C. F. Joerg, and et.al. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):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. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[8] D. Culler, S. Goldstein, and et.al. TAM – A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[9] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, 1996.

[10] J. Gurd and I. Watson. A Multi-Layered Dataflow Computer Architecture. In *International Conference on Parallel Programming -ICPP'77*, page 94, 1977.

[11] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[12] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-Based Programming. In *European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[13] N. Kobayashi. Quasi-Linear Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages pp. 29–42, 1999.

[14] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. In *ACM SIGPLAN-SIGACT Symposium on 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, University 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. In *Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, 1999.

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

[18] J. McGraw, S. Skedzielewski, and et.al. *The SISAL Language Reference Manual – Version 1.2*, 1985.

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

[20] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[21] R. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. *International Journal of High Speed Computing*, 5:171–223, 1993.

[22] R. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing. In *16th International Symposium on Computer Architecture*, pages 262–272, 1989.

[23] R. Nikhil, G. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th International Symposium on Computer Architecture*, pages 156–167, 1992.

[24] G. Papadopoulos and D. Culler. Monsoon: An Explicit Token-Store Architecture. In *17th International Symposium on Computer Architecture*, pages 82–91, 1990.

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

[26] B. Smith. A Pipelined, Shared Resource MIMD Computer. In *International Conference on Parallel Programming - ICPP'78*, pages 6–8, 1978.

[27] G. Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.

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

[29] V. Vasconcelos. Processes, Functions, Datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[30] V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, DI / FCUL, 1998.

[31] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[32] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, 1993.