A Multithreaded Typed Assembly Language*

Vasco T. Vasconcelos
Department of Informatics
Faculty of Sciences
University of Lisbon, Portugal
vv@di.fc.ul.pt

Francisco Martins
Departments of Mathematics
University of Azores, Portugal
fmartins@di.fc.ul.pt

ABSTRACT

We present an assembly language targeted at shared memory multiprocessors, where CPU cores synchronize via locks, acquired with a traditional test and set lock instruction. We show programming examples taken from the literature on Operating Systems, and discuss a typing system that enforces a strict protocol on lock usage and that prevents race conditions.

1. MOTIVATION

The need for fast information processing is one of the driving forces in the advancement of technology in general, and of computers in particular. Since the early steps in computing, when the ENIAC performed 300 operations per second, a huge strode has been made towards the computing power of nowadays machines. Nevertheless, the computing challenges have increased even faster, and the demands, for instance, from the astronomical community trying to probe the universe or from the biological community trying to understand the human genome, constantly take current computing power to the limit. What directions may we follow to increase this computing power? Olukotun and Hammond write:

> With the exhaustion of essentially all performance gains that can be achieved for "free" with technologies such as superscalar dispatch and pipelining, we are now entering an era where programmers *must* switch to more parallel programming models in order to exploit multiprocessors effectively, if they desire improved single-program performance. [17].

Continuing, we read "Previously it was necessary to minimize communication between independent threads to an extremely low level [..] Within any CMP (chip multiprocessors) with a shared on-chip cache memory, however, each communication event typically takes just a handful of processor cycles [..] Programmers must still divide their work into parallel threads, but do not need to worry nearly as much about ensuring that these threads are highly independent, since communication is relatively cheap. This is not a complete panacea, however, because programmers must still structure their inter-thread synchronization correctly, or the program may generate incorrect results or deadlock."

*This work as partially supported by the European Union under the IST Integrated Project "Software Engineering for Service-Oriented Overlay Computers".

This work is about language support to help correctly structuring inter-thread synchronization in CMPs. We design a simple abstract CMP and present its programming language: a conventional typed assembly language [16] extended with a notion of locks [6], and a fork primitive. A type system enforces a policy of lock usage, making sure that, within a thread, locks are created, locked, the shared memory accessed, and unlocked.

Our type system closely follows the tradition of typed assembly languages [14, 15, 16], extended with support for threads and locks, following Flanagan and Abadi [6]. With respect to [6], however, our work is positioned at a much lower abstraction level, and faces different challenges inherent to non-lexical scoped languages.

Lock primitives have been discussed in the context of concurrent object calculi [5], JVM [7, 8, 11, 12], C- - [19], but not in that of typed assembly languages. In a typed setting, were programs are guaranteed not to suffer from race conditions, we

- syntactically decouple of the lock and unlock operations on what one usually finds unified in a single syntactic construct in high-level languages: Birrel's *lock-do-end* construct [2], used under different names (*sync*, *synchronized-in*, *lock-in*) in a number of other works, including the Java programming language [6, 5, 7, 8, 4, 3, 9];
- allow for the lock acquisition/release in schemes other than the nested discipline imposed by the *lock-do-end* construct;
- allow to fork threads holding locks.

We describe the architecture of our CMP and its lock discipline (enforced by the type system) in Section 2. After,

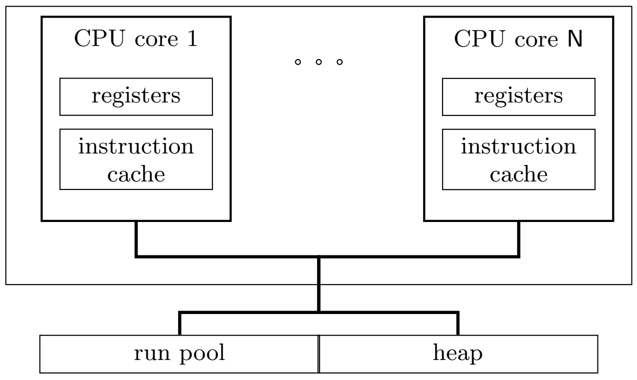

Figure 1: The architecture of the multiprocessor

we present the syntax and the operational semantics of MIL (Section 3) and sketch the programming model (Section 4), by discussing well-known examples from the literature on Operating Systems, namely the enforcement of mutual exclusion with locks. Section 5 presents a type discipline to enforce the absence of race conditions in our language. Type safety is discussed in Section 6. Some extensions to MIL and its type system are discussed in Section 7. The closing section discusses the related work, summarizes the conclusions, and points future directions to extend our research.

2. THE ARCHITECTURE OF THE MULTIPROCESSOR AND ITS LOCK USAGE POLICY

The architecture of the machine is described in Figure 1. It comprises a series of *processor cores* and a *main memory*. Each processor core owns a number of registers and an instruction cache. The main memory is divided into two parts: a conventional *heap* storing data and code, and a *run pool* storing suspended threads.

Threads run in processor cores. When the number of threads is larger than the number of available processor cores, part of the threads is placed in the run pool. For each suspended thread, the run pool stores a pair comprising (a) a pointer to the heap, where the thread's code fragment resides, and (b) a register file (a mapping from registers to values) containing the initial state of the processor.

New threads are placed in the run pool via a dedicated fork instruction. Running threads relinquish the processor by explicitly executing a yield instruction; there is no otherwise machine-wide thread suspension mechanism—our machine fits in the *cooperative thread model*, according to Tanenbaum terminology [21] (we are working at an abstraction level below that of the operating system, where one usually finds preemptive models). Freed processors look for work in the run pool. A pair is selected and removed from the pool; registers are loaded from the pair's second component (a register file), and control is transferred to the code pointed by the pair's first component (a label).

To provide for inter-thread synchronization the machine provides *locks* to protect tuples in the heap. A standard "test and set lock" instruction is used to obtain a lock, thus allowing entering a *critical region*. Running threads read and write from the shared heap via conventional load and store instructions. The policy for the usage of locks (enforced by the type system) by a given processor is depicted in Figure 2, where α denotes a *lock singleton type* and Λ the set of locks held by the processor (the processor' *permission*).

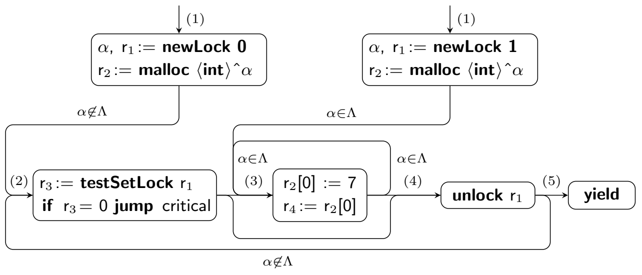

Figure 2: The Lock discipline.

registers	r	$::=$	$\mathsf{r}_1 \mid \dots \mid \mathsf{r}_\mathsf{R}$
integer values	n	$::=$	$\dots \mid$ -1 $\mid$ 0 $\mid$ 1 $\mid \dots$
lock values	b	$::=$	$\mathbf{0} \mid \mathbf{1}$
values	v	$::=$	$r \mid n \mid b \mid l \mid ?\tau$
instructions	ι	$::=$	
control flow			$r := v \mid r := r + v \mid$
			if $r = v$ jump $v \mid$
memory			$r :=$ malloc $[\vec{\tau}]$ guarded by $\alpha \mid$
			$r := v[n] \mid r[n] := v \mid$
lock			$\alpha, r :=$ newLock $b \mid$
			$r :=$ testSetLock $v \mid$ unlock $v \mid$
fork			fork v
inst. sequences	I	$::=$	$\iota; I \mid$ jump $v \mid$ yield

Figure 3: Instructions

3. SYNTAX AND OPERATIONAL SEMANTICS

The syntax of our language is described by the grammar in Figures 3, 4, and 9. We defer the treatment of types until Section 5. We rely on a set of *heap labels* ranged over by l, and a disjoint set of *type variables* ranged over by α, β. Our machine is parameterized by two values: the *number of processors* N in the machine, and the *number of registers per processor* R.

Instructions, Figure 3. Most of the proposed instructions are standard in assembly languages. Instructions are organized as *basic blocks*, that is sequences of instructions ending with a jump or a yield. The yield instruction releases the processor, allowing it to fetch another thread from the run pool.

$$\begin{array}{llcl}
\textit{lock sets} & \Lambda & ::= & \alpha_1, \ldots, \alpha_n \\
\textit{register files} & R & ::= & \{\mathsf{r}_1 \colon v_1, \ldots, \mathsf{r}_\mathsf{R} \colon v_\mathsf{R}\} \\
\textit{processor} & p & ::= & \langle R; \Lambda; I\rangle \\
\textit{processors array} & P & ::= & \{1 \colon p_1, \ldots, \mathsf{N} \colon p_\mathsf{N}\} \\
\textit{thread pool} & T & ::= & \{\langle l_1, R_1\rangle, \ldots, \langle l_n, R_n\rangle\} \\
\textit{heap values} & h & ::= & \langle v_1 \ldots v_n\rangle^\alpha \mid \tau\ \{I\} \\
\textit{heaps} & H & ::= & \{l_1 \colon h_1, \ldots, l_n \colon h_n\} \\
\textit{states} & S & ::= & \langle H; T; P\rangle \mid \mathsf{halt}
\end{array}$$

Figure 4: Abstract machine

$$\frac{\forall i. P(i) = \langle \_; \_; \mathsf{yield}\rangle}{\langle \_; \emptyset; P\rangle \to \mathsf{halt}} \qquad (\text{R-HALT})$$

$$\frac{P(i) = \langle \_; \_; \mathsf{yield}\rangle \qquad H(l) = \_\ \mathsf{requires}\ \Lambda\ \{I\}}{\langle H; T \uplus \{\langle l, R\rangle\}; P\rangle \to \langle H; T; P\{i \colon \langle R; \Lambda; I\rangle\}\rangle} \qquad (\text{R-SCHEDULE})$$

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda \uplus \Lambda'; (\mathsf{fork}\ v; I)\rangle \\ \hat{R}(v) = l \qquad H(l) = \_\ \mathsf{requires}\ \Lambda\ \{\_\} \end{array}}{\langle H; T; P\rangle \to \langle H; T \cup \{\langle l, R\rangle\}; P\{i \colon \langle R; \Lambda'; I\rangle\}\rangle} \qquad (\text{R-FORK})$$

Figure 5: Operational semantics (thread pool)

Machines, Figure 4. *Machines* can be in two states: halted or running. In the latter case, they comprise a heap, a thread pool, and a processor array. *Heaps* are maps from heap labels into *heap values* that can be either tuples or code blocks. *Tuples* are vectors of values protected by some lock α (locks play no role at runtime; they are needed only for Subject Reduction and Type Safety). *Code blocks* comprise a signature and a body: the *signature*, which the type system makes sure is of the form Γ requires Λ, describes the types of each register and the locks held by the processor when jumping to the block code; the *body* is a sequence of instructions.

A *thread pool* is a multiset of pairs, each of which contains a pointer to a code block and a register file. A *processor array* contains N processors (recall that N is a parameter to the machine). Each *processor* is composed of a register file (of fixed length R, the other parameter to the machine), a set of locks (the locks held by the thread running in the processor), and a sequence of instructions.

The operational semantics is defined via a reduction relation on machine states, as described in Figures 5 to 8.

Thread pool instructions, Figure 5. Rule R-HALT stops the machine when it finds an empty thread pool and all processors idle. Otherwise, if there is an idle processor and a pair in the thread pool, then rule R-SCHEDULE assigns a new thread to the processor. A pair label-registers is taken from the pool; the instructions for, and the locks held by, the new thread are read from the code block addressed by the label; the initial value of the registers are read from the pair. Rule R-FORK places a new thread in the thread pool. Notice that part of the held locks go with the forked thread, while the rest remains in the thread.

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{newLock}\ \mathbf{0}; I)\rangle \qquad l \notin \mathrm{dom}(H), \beta \text{ fresh}}{\langle H; T; P\rangle \to \langle H\{l \colon \langle \mathbf{0}\rangle^\beta\}; T; P\{i \colon \langle R\{r \colon l\}; \Lambda; I[\beta/\alpha]\rangle\}\rangle} \qquad (\text{R-NEWLOCK } \mathbf{0})$$

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{newLock}\ \mathbf{1}; I)\rangle \qquad l \notin \mathrm{dom}(H), \beta \text{ fresh}}{\langle H; T; P\rangle \to \langle H\{l \colon \langle \mathbf{1}\rangle^\beta\}; T; P\{i \colon \langle R\{r \colon l\}; \Lambda \uplus \{\beta\}; I[\beta/\alpha]\rangle\}\rangle} \qquad (\text{R-NEWLOCK } \mathbf{1})$$

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{testSetLock}\ v; I)\rangle \qquad \hat{R}(v) = l \qquad H(l) = \langle \mathbf{0}\rangle^\alpha}{\langle H; T; P\rangle \to \langle H\{l \colon \langle \mathbf{1}\rangle^\alpha\}; T; P\{i \colon \langle R\{r \colon \mathbf{0}\}; \Lambda \uplus \{\alpha\}; I\rangle\}\rangle} \qquad (\text{R-TSL } \mathbf{0})$$

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{testSetLock}\ v; I)\rangle \qquad H(\hat{R}(v)) = \langle \mathbf{1}\rangle^\alpha}{\langle H; T; P\rangle \to \langle H; T; P\{i \colon \langle R\{r \colon \mathbf{1}\}; \Lambda; I\rangle\}\rangle} \qquad (\text{R-TSL } \mathbf{1})$$

$$\frac{P(i) = \langle R; \Lambda \uplus \{\alpha\}; (\mathsf{unlock}\ v; I)\rangle \qquad \hat{R}(v) = l \qquad H(l) = \langle \_\rangle^\alpha}{\langle H; T; P\rangle \to \langle H\{l \colon \langle \mathbf{0}\rangle^\alpha\}; T; P\{i \colon \langle R; \Lambda; I\rangle\}\rangle} \qquad (\text{R-UNLOCK})$$

Figure 6: Operational semantics (locks)

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{malloc}\ [\vec{\tau}]\ \mathsf{guarded\ by}\ \alpha; I)\rangle \qquad l \notin \mathrm{dom}(H)}{\langle H; T; P\rangle \to \langle H\{l \colon \langle \vec{?\tau}\rangle^\alpha\}; T; P\{i \colon \langle R\{r \colon l\}; \Lambda; I\rangle\}\rangle} \qquad (\text{R-MALLOC})$$

$$\frac{P(i) = \langle R; \Lambda; (r := v[n]; I)\rangle \qquad H(\hat{R}(v)) = \langle v_1..v_n..v_{n+m}\rangle^\alpha}{\langle H; T; P\rangle \to \langle H; T; P\{i \colon \langle R\{r \colon v_n\}; \Lambda; I\rangle\}\rangle} \qquad (\text{R-LOAD})$$

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda; (r[n] := v; I)\rangle \\ R(r) = l \qquad H(l) = \langle v_1..v_n..v_{n+m}\rangle^\alpha \end{array}}{\langle H; T; P\rangle \to \langle H\{l \colon \langle v_1..\hat{R}(v)..v_{n+m}\rangle^\alpha\}; T; P\{i \colon \langle R; \Lambda; I\rangle\}\rangle} \qquad (\text{R-STORE})$$

Figure 7: Operational semantics (memory)

Lock instructions, Figure 6. The newLock instructions create new locks, in a locked or unlocked state. The scope of α is the rest of the code block. A tuple—$\langle \mathbf{0}\rangle^\beta$ in the former case; $\langle \mathbf{1}\rangle^\beta$ in the latter—is allocated in the heap, and register r is made to point to it. A fresh type variable β replaces the variable α chosen by the programmer. When the lock is created in the locked state, the new lock variable β is added to the set of the locks held by the processor. Locks apart, an instruction $\alpha, r := \mathsf{newLock}\ b$ behaves as the pair of instructions $r := \mathsf{malloc}\ \langle \mathsf{lock}(\alpha)\rangle^\alpha; r[0] := b$. Instruction $r := \mathsf{testSetLock}\ v$ is the *Test and Set Lock* present in many machines designed with multiple processes in mind. It reads the contents of the memory word v into register r and then stores $\mathbf{1}$ at the memory address v. When a locked lock is found at the memory address, its lock variable α is added to the permissions of the processor. As usual, the two operations (load and store) are indivisible—no other processor can access the memory word until the instruction is finished; our operational semantics enforces this behavior. Locks apart, an instruction $\mathsf{unlock}\ v$ behaves as $r[0] := \mathbf{0}$. Rule R-UNLOCK, however, makes sure the processor holds the lock.

$$\frac{P(i) = \langle R; \Lambda; \mathsf{jump}\ v\rangle \qquad H(\hat{R}(v)) = \_\ \{I\}}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle} \quad (\text{R-JUMP})$$

$$\frac{P(i) = \langle R; \Lambda; (r := v; I)\rangle}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon \hat{R}(v)\}; \Lambda; I\rangle\}\rangle} \quad (\text{R-MOVE})$$

$$\frac{P(i) = \langle R; \Lambda; (r := r' + v; I)\rangle}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon R(r') + \hat{R}(v)\}; \Lambda; I\rangle\}\rangle} \quad (\text{R-ARITH})$$

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda; (\mathsf{if}\ r = v\ \mathsf{jump}\ v'; \_)\rangle \\ R(r) = v \qquad H(\hat{R}(v')) = \_\ \{I\} \end{array}}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle} \quad (\text{R-BRANCHT})$$

$$\frac{P(i) = \langle R; \Lambda; (\mathsf{if}\ r = v\ \mathsf{jump}\ \_; I)\rangle \qquad R(r) \neq v}{\langle H; T; P\rangle \to \langle H; T; \{i\colon \langle R; \Lambda; I\rangle\}\rangle} \quad (\text{R-BRANCHF})$$

Figure 8: Operational semantics (control flow)

Memory instructions, Figure 7. The rule for malloc allocates a new tuple in the heap, protected by a given lock, and makes register r point to it. The size of the tuple is that of sequence of types $[\vec{\tau}]$, its values are uninitialized values. The rules for loading and storing are standard [14].

Control flow instructions, Figure 8. These transition rules are mostly straightforward [14]. They rely on the *eval* function that works on *operands* (that is, registers or values), by looking for values in registers.

$$\hat{R}(v) = \begin{cases} R(v) & \text{if } v \text{ is a register} \\ v & \text{otherwise} \end{cases}$$

4. INTERPROCESSOR COMMUNICATION

This section presents the main concepts of our language, based on classical examples taken from the literature on Operating Systems [21]. The following examples illustrate the usage of threads and the discipline of *locks.*

4.1 Mutual exclusion using busy waiting

We start with a code block that actively tries to acquire a lock. This technique, called *spin lock*, is used when there is a reasonable expectation that the lock will be available in a short period of time.

Let Tuple stands for type $\langle \textbf{int}\rangle\hat{}\alpha$, the type of tuples with one integer component, protected by lock α. In order to read or to write values in a heap location of type Tuple, the thread must hold lock α. In this example, we allocate a tuple, acquire the lock, and store a value in the tuple (within a critical region where we hold the lock).

The below code block actively tries to acquire lock α (supplied in register r_2); on success it transfers control to a critical region. The code block expects the (address of the) tuple in register r_1. The tuple plays no rôle in the code block; it is only intended to be passed to the critical region.[1]

```
enterSpinLockRegion(r1: Tuple, r2: ⟨lock(α)⟩^α) {
  r3 := testSetLock r2     -- try to acquire lock α
  if r3 = 0 jump criticalRegion   -- if so ...
  jump enterSpinLockRegion   -- else wait
}
```

A critical region stores a value in the tuple, releases the lock, and terminates. Notice that the type for the critical region expects the current thread to hold lock α, as indicated by α after the semicolon in the signature of the code block. After updating the tuple, lock α is released, marking the end of the critical region.

```
criticalRegion(r1: Tuple, r2: ⟨lock(α)⟩^α)
    requires α {
  r1[0] := 10            -- update the tuple
  unlock r2              -- release the lock
  jump continuation      -- continue
}
```

Finally, the main code block creates a new (unlocked) lock, allocates a tuple in the heap, and tries to acquire lock α by transferring control to enterSpinLockRegion.

```
main () {
  α, r2 := newLock 0          -- create µtex lock
  r0 := malloc [int]^α        -- allocate tuple
  jump enterSpinLockRegion -- try acquire lock
}
```

4.2 Mutual exclusion using threads

We now discuss a distinct, yet classical, approach to lock acquisition. The idea is to avoid actively waiting for the lock, by launching a thread that tries to acquire the lock. If it succeeds, control is transferred to the critical region. Otherwise, the thread forks another thread that tries to gather the lock later, thus avoiding a busy wait.

The code and type the for criticalRegion is that of the previous example. The code for main is identical, apart from the last instruction that is replaced by **jump** enterSleepLockRegion. We discuss how to acquire lock α by forking a new thread.

To fork a thread we use instruction **fork** and specify the label of the code block that should be run when a processor is available. Upon thread starting, registers are loaded with the values they had when the fork action happened. In the present case, register r_1 should contain the address of the tuple and register r_2 the lock to be acquired.

When enterSleepLockRegion fails to acquire the lock, it creates a new thread (that will try to obtain the lock later) and terminates the current one, rather than jumping to the beginning of the code block, as it happens with enterSpinLockRegion.

```
enterSleepLockRegion (r1: Tuple, r2: ⟨lock(α)⟩^α) {
 r3 := testSetLock r2        -- try to acquire lock
 if r3 = 0 jump criticalRegion -- if so, ...
```

[1]In order to make code blocks reusable in different contexts, we need to abstract the type of the tuple. For the sake of simplicity and clarity, in this paper we do not make use of existential types. The interested reader may refer to [13] for examples illustrating continuation passing style using existential types.

types	τ	$::=$	$\mathsf{int} \mid \langle\vec{\sigma}\rangle^{\alpha} \mid \Gamma \text{ requires } \Lambda \mid \mathsf{lock}(\alpha)$
init types	σ	$::=$	$\tau \mid ?\tau$
register file types	Γ	$::=$	$\mathsf{r}_1 : \tau_1, \ldots, \mathsf{r}_n : \tau_n$
typing environment	Ψ	$::=$	$\emptyset \mid \Psi, l \colon \tau \mid \Psi, \alpha :: \mathsf{Lock}$

Figure 9: Types

$$\vdash \langle\sigma_1, \ldots, \tau_n, \ldots, \sigma_{n+m}\rangle^{\alpha} <: \langle\sigma_1, \ldots, ?\tau_n, \ldots, \sigma_{n+m}\rangle^{\alpha} \qquad \text{(S-UNINIT)}$$

$$\vdash \sigma <: \sigma \qquad \frac{\vdash \sigma <: \sigma' \qquad \vdash \sigma' <: \sigma''}{\vdash \sigma <: \sigma''} \qquad \text{(S-REF, S-TRANS)}$$

$$\frac{\vdash \tau' <: \tau}{\Psi, l \colon \tau' \vdash l \colon \tau} \qquad \Psi \vdash n \colon \mathsf{int} \qquad \Psi \vdash b \colon \mathsf{lock}(\alpha) \qquad \Psi \vdash ?\tau \colon ?\tau$$

(T-INT,T-LABEL,T-LOCK,T-UNINIT)

$$\Psi; \Gamma \vdash r \colon \Gamma(r) \qquad \frac{\Psi \vdash v \colon \tau}{\Psi; \Gamma \vdash v \colon \tau} \qquad \text{(T-REG,T-VAL)}$$

Figure 10: Typing rules for values $\boxed{\Psi \vdash v \colon \tau}$ and for operands $\boxed{\Psi; \Gamma \vdash v \colon \tau}$

$$\Psi; \Gamma; \emptyset \vdash \mathsf{yield} \qquad \text{(T-YIELD)}$$

$$\frac{\Psi; \Gamma \vdash v \colon \Gamma \text{ requires } \Lambda \qquad \Psi; \Gamma; \Lambda' \vdash I}{\Psi; \Gamma; \Lambda \uplus \Lambda' \vdash \mathsf{fork}\ v; I} \qquad \text{(T-FORK)}$$

$$\frac{\Psi, \alpha :: \mathsf{Lock}; \Gamma\{r \colon \langle\mathsf{lock}(\alpha)\rangle^{\alpha}\}; \Lambda \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha, r := \mathsf{newLock}\ \mathbf{0}; I}$$

(T-NEWLOCK **0**)

$$\frac{\Psi, \alpha :: \mathsf{Lock}; \Gamma\{r \colon \langle\mathsf{lock}(\alpha)\rangle^{\alpha}\}; \Lambda \uplus \{\alpha\} \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha, r := \mathsf{newLock}\ \mathbf{1}; I}$$

(T-NEWLOCK **1**)

$$\frac{\Psi; \Gamma \vdash v \colon \langle\mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \Psi; \Gamma\{r \colon \mathsf{lock}(\alpha)\}; \Lambda \vdash I \qquad \alpha \notin \Lambda}{\Psi; \Gamma; \Lambda \vdash r := \mathsf{testSetLock}\ v; I}$$

(T-TSL)

$$\frac{\Psi; \Gamma \vdash v \colon \langle\mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \Psi; \Gamma; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \uplus \{\alpha\} \vdash \mathsf{unlock}\ v; I} \qquad \text{(T-UNLOCK)}$$

$$\frac{\Psi; \Gamma \vdash r \colon \mathsf{lock}(\alpha) \quad \Psi; \Gamma \vdash v \colon \Gamma \text{ requires } (\Lambda \uplus \{\alpha\}) \quad \Psi; \Gamma; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash \mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I}$$

(T-CRITICAL)

Figure 11: Typing rules for instructions (thread pool and locks) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$

```
  fork enterSleepLockRegion  -- else, try later
  yield                      -- free the processor
}
```

Conventional pthread mutex implementations maintain a queue of waiting threads, rather than repeatedly forking threads. Section 7.3 sketches such an implementation.

5. TYPE DISCIPLINE

The syntax of types is described in Figure 9. A type of the form $\langle\vec{\sigma}\rangle^{\alpha}$ describes a tuple in the heap protected by lock α. Each type in $\vec{\sigma}$ is either initialized (τ) or uninitialized ($?\tau$). A type of the form Γ requires Λ describes a code block; a thread jumping into such a block must hold a register file type Γ as well as the locks in Λ. The type $\mathsf{lock}(\alpha)$ describes a singleton lock type.

The type system is defined in Figures 10 to 13.

Typing values, Figure 10. Heap values are distinguished from operands (that include registers as well) by the form of the sequent. Notice that lock values—$\mathbf{0}, \mathbf{1}$—have any lock type. Also, an uninitialized value $?\tau$ has a type $?\tau$; we use the same syntax for a uninitialized value (at the left of the colon) and its type (at the right of the colon). A formula $\sigma <: \sigma'$ allows to "forget" initializations.

Typing fork and lock instructions, Figure 11. Instructions are checked against a typing environment Ψ (mapping labels to types, and type variables to the kind Lock: the kind of singleton lock types), a register file type Γ holding the current types of the registers, and a set Λ of lock variables (the *permission* of the code block).

Rule T-YIELD requires an empty permission meaning that all locks must have been released prior to ending the thread. Only the thread that acquired a lock may release it (see rule T-UNLOCK below); as such, allowing acquired locks to "die" with the thread, may lead to deadlock situations (cf. [11]).

Rule T-FORK splits the permission into two sets, Λ and Λ': one goes with the forked thread, the other remains with the current thread. Such a scheme is crucial in the implementation of Hoare's monitors [10], as described in Section 7.3.

The two rules for newLock assign a lock type to the register. When the lock is created in the locked state, the singleton type is added to the permission of the thread. Rule T-TSL requires that the value under test holds a lock; disallowing testing a lock already held by the thread. Rule T-UNLOCK makes sure that only held locks are unlocked. Finally, the jump-to-critical-region rule ensures that the current thread holds the exact number of locks required by the target code block. The rule also adds the lock under test to the permission of the thread. A thread is guaranteed to hold the lock only after (conditionally) jumping to a critical region. A previous test and set lock instruction may have obtained the lock, but as far as the type system goes, the thread holds the lock after a jump-to-critical-region instruction.

Typing memory and control instructions, Figure 12. These rules are standard [14], except on what concerns the locks. The rule for malloc makes sure the lock α is in scope, meaning that it must be preceded by a $\alpha, r := \mathsf{newLock}\ b$ instruction, in the *same* code block; Section 7.2 shows how to overcome this limitation. The rules for load and store make sure that lock α is in the permission Λ of the thread. The conditions regarding lock type $\mathsf{lock}(\_)$ in rules T-MALLOC, T-LOAD, and T-STORE ensure that locks are only created using a newLock instruction, and manipulated with test and set lock.

$$\frac{\Psi, \alpha :: \mathsf{Lock}; \Gamma\{r \colon \langle ?\vec{\tau}\rangle^{\alpha}\}; \Lambda \vdash I \qquad \vec{\tau} \neq \mathsf{lock}(\_)}{\Psi, \alpha :: \mathsf{Lock}; \Gamma; \Lambda \vdash r := \mathsf{malloc}\ [\vec{\tau}]\ \mathsf{guarded\ by}\ \alpha; I} \quad (\text{T-MALLOC})$$

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash v \colon \langle \sigma_1..\tau_n..\sigma_{n+m}\rangle^{\alpha} \qquad \Psi; \Gamma\{r \colon \tau_n\}; \Lambda \vdash I \\ \tau_n \neq \mathsf{lock}(\_) \qquad \alpha \in \Lambda\end{array}}{\Psi; \Gamma; \Lambda \vdash r := v[n]; I} \quad (\text{T-LOAD})$$

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash v \colon \tau_n \quad \Psi; \Gamma \vdash r \colon \langle \sigma_1..\sigma_n..\sigma_{n+m}\rangle^{\alpha} \quad \tau_n \neq \mathsf{lock}(\_) \\ \Psi; \Gamma\{r \colon \langle \sigma_1..\mathrm{type}(\sigma_n)..\sigma_{n+m}\rangle^{\alpha}\}; \Lambda \vdash I \quad \alpha \in \Lambda\end{array}}{\Psi; \Gamma; \Lambda \vdash r[n] := v; I} \quad (\text{T-STORE})$$

$$\frac{\Psi; \Gamma \vdash v \colon \tau \qquad \Psi; \Gamma\{r \colon \tau\}; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash r := v; I} \quad (\text{T-MOVE})$$

$$\frac{\Psi; \Gamma \vdash r' \colon \mathsf{int} \qquad \Psi; \Gamma \vdash v \colon \mathsf{int} \qquad \Psi; \Gamma\{r \colon \mathsf{int}\}; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash r := r' + v; I} \quad (\text{T-ARITH})$$

$$\frac{\Psi; \Gamma \vdash r \colon \mathsf{int} \qquad \Psi; \Gamma \vdash v \colon \Gamma\ \mathsf{requires}\ \Lambda \qquad \Psi; \Gamma; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash \mathsf{if}\ r = 0\ \mathsf{jump}\ v; I} \quad (\text{T-BRANCH})$$

$$\frac{\Psi; \Gamma \vdash v \colon \Gamma\ \mathsf{requires}\ \Lambda}{\Psi; \Gamma; \Lambda \vdash \mathsf{jump}\ v} \quad (\text{T-JUMP})$$

where $\mathrm{type}(\tau) = \mathrm{type}(?\tau) = \tau$.

Figure 12: Typing rules for instructions (memory and control flow) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$

Typing machine states, Figure 13. The rules should be easy to understand. The only remark goes to heap tuples, where we make sure that all locks protecting the tuples are in the domain of the typing environment.

The main result of this section follows.

THEOREM 1 (SUBJECT REDUCTION). *If $\vdash S$ and $S \to S'$, then $\vdash S'$.*

6. TYPE SAFETY

We split the results in three categories: the standard "well-typed machines do not get stuck" (which we omit altogether), the lock discipline, and races. The lock discipline is embodied in the following Lemma (cf. Figure 2).

LEMMA 2 (LOCK DISCIPLINE). *Let $\vdash H \colon \Psi$ and $\Psi \vdash \langle R; \Lambda; (\iota; \_)\rangle$.*

1. *Before lock creation, α is not a known lock. If ι is $\alpha, \_ :=$ newLock $\_$, then $\alpha \notin \mathrm{dom}(\Psi)$.*
2. *Before test and set lock, the processor does not hold the lock. If ι is $\_ :=$ testSetLock v and $H(\hat{R}(v)) = \langle \mathit{lock}(\alpha)\rangle^{-}$, then $\alpha \notin \Lambda$.*
3. *Before accessing the heap, the processor holds the lock. If ι is $v[\_] := \_$ or $\_ := v[\_]$, and $H(\hat{R}(v)) = \langle \_\rangle^{\alpha}$, then $\alpha \in \Lambda$.*

$$\frac{\forall i. \Psi \vdash R(r_i) \colon \Gamma(r_i)}{\Psi \vdash R \colon \Gamma} \quad (\text{reg file, } \boxed{\Psi \vdash R \colon \Gamma})$$

$$\frac{\forall i. \Psi \vdash P(i)}{\Psi \vdash P} \qquad \frac{\Psi \vdash R \colon \Gamma \qquad \Psi; \Gamma; \Lambda \vdash I}{\Psi \vdash \langle R; \Lambda; I\rangle} \quad (\text{processors, } \boxed{\Psi \vdash P})$$

$$\frac{\forall i. \Psi \vdash l_i \colon \Gamma_i\ \mathsf{requires}\ \_ \qquad \Psi \vdash R_i \colon \Gamma_i}{\Psi \vdash \{\langle l_1, R_1\rangle, \ldots, \langle l_n, R_n\rangle\}} \quad (\text{thread pool, } \boxed{\Psi \vdash T})$$

$$\frac{\Psi; \Gamma; \Lambda \vdash I}{\Psi \vdash \Gamma\ \mathsf{requires}\ \Lambda\ \{I\} \colon \tau} \qquad \frac{\forall i. \Psi, \alpha :: \mathsf{Lock} \vdash v_i \colon \tau_i}{\Psi, \alpha :: \mathsf{Lock} \vdash \langle \vec{v}\rangle^{\alpha} \colon \langle \vec{\sigma}\rangle^{\alpha}} \quad (\text{heap value, } \boxed{\Psi \vdash h \colon \tau})$$

$$\frac{\forall l. \Psi \vdash H(l) \colon \Psi(l)}{\Psi \vdash H} \quad (\text{heap, } \boxed{\Psi \vdash H})$$

$$\vdash \mathsf{halt} \qquad \frac{\Psi \vdash H \qquad \Psi \vdash T \qquad \Psi \vdash P}{\vdash \langle H; T; P\rangle} \quad (\text{state, } \boxed{\vdash S})$$

Figure 13: Typing rules for machine states

4. *Unlock only in possession of the lock. If ι is unlock v and $H(\hat{R}(v)) = \langle \_\rangle^{\alpha}$, then $\alpha \in \Lambda$.*
5. *Releasing the processor only without held locks. If ι is yield, then $\Lambda = \emptyset$.*

For races we follow Flanagan and Abadi [6]. We start by defining the *set of permissions* of a machine state, by joining the permissions of the processes with those of the threads in the run pool, and with the set of unlocked locks in the heap. Remember that a permission is a set of locks, denoted by Λ.

DEFINITION 1 (STATE PERMISSIONS.).

$$\begin{aligned}
\mathcal{L}_P &= \{\Lambda \mid P(i) = \langle \_; \Lambda; \_\rangle\} \\
\mathcal{L}_T &= \{\Lambda \mid \langle l, \_\rangle \in T \text{ and } H(l) = \_\ \mathit{requires}\ \Lambda\ \{\_\}\} \\
\mathcal{L}_H &= \{\{\alpha \mid H(l) = \langle \mathit{0}\rangle^{\alpha}\}\} \\
\mathcal{L}_{\langle H;T;P\rangle} &= \mathcal{L}_P \cup \mathcal{L}_T \cup \mathcal{L}_H \\
\mathcal{L}_{\mathit{halt}} &= 2^{2^{L}}
\end{aligned}$$

State permissions do not shrink with reduction. The proof (a case analysis on the various reduction rules) shows that state permissions grow only due to the execution of a newLock instruction.

LEMMA 3. *If $S \to S'$, then $\mathcal{L}_S \subseteq \mathcal{L}_{S'}$.*

We are interested only in *mutual exclusive states*, that is, states whose permissions do not "overlap." Also, we say, that a state has a *race condition* if it contains two processors trying to access the heap at the same location.

DEFINITION 2. **Mutual exclusive states.** *halt is mutual exclusive; $S \neq$ halt is* mutual exclusive *when $i \neq j$ implies $\Lambda_i \cap \Lambda_j = \emptyset$, for all $\Lambda_i, \Lambda_j \in \mathcal{L}_S$.*

$$\textit{types} \quad \tau \ ::= \ \ldots \ | \ \mu X.\tau \ | \ X$$

Figure 14: Extending the type system with recursive types

Accessing the heap. *A processor of the form* $\langle R; \_; (\iota; \_)\rangle$ accesses heap H at location l, *if* ι *is of the form* $v[\_] := \_$ *or of the form* $\_ := v[\_]$, *and* $l = H(\hat{R}(v))$.

Race condition. *A state* S *has a* race condition *if* $S = \langle H; \_; P\rangle$ *and there exist* i *and* j *distinct such that* $P(i)$ *and* $P(j)$ *both access heap* H *at some location* l.

THEOREM 4 (TYPES AGAINST RACES). *If* $\vdash S$ *and* S *is mutual exclusive, then* S *does not have a race condition.*

7. EXTENSIONS

In this section we discuss the extensions to the language required to implement Hoare's monitors [10].

7.1 Recursive types

Condition variables in monitors are represented by queues. Queues are usually represented by lists, where each node refers to the next node in the list, yielding a recursive data structure.

Extending the language to include recursive types is straightforward. Figure 14 summarizes the changes to the syntax. The μ operator is a binder, giving rise, in the standard way, to notions of bound and free variables and alpha-equivalence. We do not distinguish between alpha-convertible types. Furthermore, we take an equi-recursive view of types [18], not distinguishing between a type $\mu X.\tau$ and its unfolding $\tau[\mu X.\tau/X]$.[2]

7.2 Polymorphism over lock types

We would like to protect the queue, and all the nodes in it with the monitor's lock. The problem is that scope of a lock ranges from its creation to the end of the code block where it was created, disallowing manipulating the condition (for example with the wait and signal primitives) in distinct code blocks.

Polymorphism over lock types allows writing code blocks parametric on the locks they require, an in particular makes it possible to protect a heap tuple with an abstracted lock. This is particularly useful for algorithms that prefer to protect all nodes of a list with the same lock, rather than using a different lock for each node, as we show in the next section. This extension makes it possible to protect new heap tuples with a lock created in a different code block.

[2]For recursive code blocks, Morrisett *et al.* [14] introduced a mechanism allowing to "forget" register types when entering a code block. Using this technique and by carefully choosing the register to hold the continuation address one may avoid using recursive types for recursive procedures. Recursive types, however, come into play in the presence of recursive datatypes.

EXTENDED SYNTAX

$$\begin{array}{lll} \textit{values} & v ::= & \ldots \ | \ v[\vec{\alpha}] \\ \textit{types} & \tau ::= & \ldots \ | \ \forall[\vec{\alpha}].(\Gamma \ \mathsf{requires}\ \Lambda) \ | \ \ldots \end{array}$$

ADDITIONAL RULES

$$\frac{\Psi; \Gamma \vdash v \colon \forall[\vec{\alpha}].(\Gamma' \ \mathsf{requires}\ \Lambda)}{\Psi; \Gamma \vdash v[\vec{\beta}] \colon \forall[].(\Gamma'[\vec{\beta}/\vec{\alpha}] \ \mathsf{requires}\ \Lambda[\vec{\beta}/\vec{\alpha}])} \quad \text{(T-VAL-APP)}$$

CHANGED RULES

$$\frac{\Psi; \Gamma \vdash v \colon \forall[].(\Gamma \ \mathsf{requires}\ \Lambda)}{\Psi; \Gamma; \Lambda \vdash \mathsf{jump}\ v} \quad \text{(T-JUMP)}$$

$$\frac{\Psi, \vec{\alpha} :: \mathsf{Lock}; \Gamma; \Lambda \vdash I}{\Psi \vdash \Gamma \ \mathsf{requires}\ \Lambda\ \{I\} \colon \forall[\vec{\alpha}].(\Gamma \ \mathsf{requires}\ \Lambda)} \quad \text{(heap value)}$$

Rules T-FORK, T-BRANCH, and T-CRITICAL undergo changes similar to rule T-JUMP.

Figure 15: Extending the simple type system with universal types

The extension follows Morrisett *et al.* [14], and is described in Figure 15, where we omit the obvious syntactic adjustment to the reduction rules. The interesting facts to notice is that when forking or jumping to a code block (*e.g.* rule T-JUMP) all lock variables must have been instantiated using value application $v[\vec{\alpha}]$; and that, when typing a code block (rule heap value), the abstracted lock types $\vec{\alpha}$ are added to the typing environment Ψ, and may then be used to protect heap tuples (cf. rule R-MALLOC in Figure 12).

7.3 Hoare's Monitors

We focus on the implementation of conditions, in particular the primitives for creation of a new condition, wait, and signal.

A condition is represented as an (initially empty) queue of closures representing the threads that are currently waiting on the condition. To simplify the example, we use the code address only, omitting the environment. The queue is protected by the monitor's lock, which we call m. Type Condition is ⟨**int**, ⟨**lock**(m)⟩^m, Node, Node⟩^m, where the first component is the size of the queue, thereafter is the monitor's lock, and finally two references for the head and tail of the queue. Each node is a pair formed by a **Code** (that we leave unspecified) and a reference for the next node in the queue, yielding the type μ x. ⟨**Code**, X⟩^m.

We adopt a Continuation-Passing Style [1], meaning that code blocks are passed the address of the continuation a register. The following code block accepts in register r_1 the monitor's lock, in register r_2 the continuation, and requires that the thread holds lock m. It then creates a new condition variable and passes it to the continuation in register r_3. Notice that the code block is parametric in lock m, allowing to protect the queue's descriptor and sentinel node with lock m.

```
newCondition ∀[m].(r1:⟨lock(m)⟩^m, r2:Code)
    requires m {
  -- allocate the sentinel
  r4 := malloc [Closure,Node] guarded by m
  -- allocate the queue header
  r3 := malloc [int,⟨lock(m)⟩^m,Node,Node]
        guarded by m
  r3[0] := 0; r3[1] := r1 -- store size and lock
  r3[2] := r4; r3[3] := r4 -- store head and tail
  jump r2                  -- jump to continuation
}
```

The wait operation is issued from inside a monitor (represented by requiring lock m in the code block type for the operation) and causes the calling program to be delayed until a signal operation occurs. Waiting on a condition amounts to enqueue the continuation of the wait operation (represented by the **Code** type), needed when the signaling operation occurs. The condition is passed in register r_1 and the continuation in register r_2. Apart from queue manipulation details, it is important to notice that the newly created node is protected by lock m, that the lock is released, allowing other monitor procedures to execute, and that the current thread terminates.

```
wait ∀[m](r1:Condition, r2:Code) requires m {
  -- allocate a new sentinel
  r3 := malloc [Closure,Node] guarded by m
  -- update the current sentinel
  r4 := r1[3]      -- get the current sentinel
  r4[0] := r2      -- fill the continuation
  r4[1] := r3      -- fill the sentinel
  r1[3] := r3      -- store the new sentinel
  -- increment queue size
  r3 := r1[0]; r3 := r3+1; r1[0] := r3
  -- release monitor lock and terminate thread
  r3 := r1[2]; unlock r3
  yield
}
```

A signal operation, also issued from inside a monitor, causes exactly one of the waiting programs to resume immediately. A signal operation must be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program [10]. The code block for signal receives the condition in register r_1 and the continuation in register r_2. If the queue is empty, the signal has no effect. Otherwise, queue manipulation details apart, the key feature to focus our attention is in the **fork** r_3 instruction, where we crucially make use of the splitting lock mechanism when launching a new thread. In this way we are able to pass the lock to the new thread without unlocking it first. After the fork instruction the current thread no longer has lock m.

```
signal ∀[m](r1:Condition, r2:Code) requires m {
  r3 := r3[0]
  if r3 = 0 jump unlockAndGo
  -- dequeue
  r3 := r1[2]; r4 := r3[2]; r1[2] := r4
  -- decrement queue size
  r4 := r1[0]; r4 := r4 - 1; r1[0] := r4
  r3 := r3[0] -- restore the waiting thread
  fork r3     -- launch a thread inheriting m!
  jump r2     -- jump continuation (no lock m)
}
```

```
unlockAndGo ∀[m](r1:Condition, r2:Code)
    requires m {
  unlock r1[1]
  jump r2
}
```

7.4 Existential quantification over lock types

The introduction of universal types over locks allows for the construction of intricate data structures where part of the nodes may be protected by different locks, while others may share locks. However, this extra facility in lock manipulation is useful as long as locks are passed around between code blocks, when jumping or forking. Unfortunately, the technique becomes impracticable if we need to propagate locks through successive code blocks in order to recover them later, specially if the intermediate code blocks do not use the locks. An alternative is to store locks in the heap and recover them later, but the language offers no facility to store and recover singleton types from the heap.

For instance, in the monitors example, storing the singleton type m in the queue itself allows different code blocks to retrieve the lock type and enqueue a node. In fact, this code block does not care which singleton lock type protects the queue, it just requires that there exists such a lock. This example motivates the existential quantification over lock types It is straightforward to incorporate existential types in our type system, by following Flanagan and Abadi [6]. Actually, we need distinct primitives to deal with existential lock types and conventional existential types (for instance to be use with CPS). The reason is that after unpacking a value, its witness type remains abstracted, so that we can not discriminate packed lock types from other types, and hence can not use lock operations on the witness type.

8. CONCLUSION AND FURTHER WORK

We presented a typed assembly language suited for an architecture where multiple CPU cores share a common memory. The language primitively includes instructions for handling locks (create, acquire, and release) and for forking threads. We provide a type system that verifies the usage of labels, values, and registers according to the declared types and ensures a discipline on lock usage. Further results include a guarantee that well-typed states do not "get stucked" and do not incur in race conditions.

A compiler prototype can be found in URL [13], together with several examples, including extended versions of those found in this paper. During the course of program development with our language, we have faced several difficulties that should be addressed:

Exclusive-lock and shared-lock. It would be desirable to have two levels of locking: exclusive locking does not allow for any readings; shared-locking permit multiple readers. We think it could be possible to incorporate this requirements using two set of held locks (held exclusive and held shared) in the type system, adjust conveniently rules T-LOAD and T-STORE in Figure 12, having two test and set lock primitives (for exclusive and shared locking), and using three-valued locks in the operation semantics.

Avoid protecting every tuple in the heap. There are a number of situations where tuples do not require locking (for example, closures in the Continuation Passing Style [1]). A possible approach is to treat closures as some sort of linear types and try to follow the ideas expressed in [20]. Another possible direction is ownership types, where owned objects are syntactically distinguished [7, 3, 4].

Acknowledgments. The authors would like to thank the anonymous referees for valuable comments and pointers.

9. REFERENCES

[1] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[2] A. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, Palo Alto, California, 1989.

[3] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in java programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[4] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[5] C. Flanagan and M. Abadi. Object types against races. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 288–303. Springer-Verlag, 1999.

[6] C. Flanagan and M. Abadi. Types for safe locking. In *European Symposium on Programming*, volume 1576 of *LNCS*, pages 91–108. Springer-Verlag, 1999.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[8] C. Flanagan and S. N. Freund. Type inference against races. In *International Conference on Concurrency Theory*, volume 3148 of *LNCS*, pages 116–132. Springer-Verlag, 2004.

[9] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation*, volume 38(3) of *SIGPLAN Notices*, pages 13–25. ACM Press, 2003.

[10] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[11] F. Iwama and N. Kobayashi. A new type system for JVM lock primitives. In *ASIA-PEPM '02: Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–82, New York, NY, USA, 2002. ACM Press.

[12] C. Laneve. A type system for JVM threads. *Theor. Comput. Sci.*, 290(1):741–778, 2003.

[13] Mil: A multithreaded typed assembly language. `http://labmol.di.fc.ul.pt/mil/`.

[14] G. Morrisett. Typed assembly language. In *Advanced Topics in Types and Programming Languages.* MIT press, 2005.

[15] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for Systems Software*, Atlanta, May 1999.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[17] K. Olukotun and L. Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–34, 2005.

[18] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[19] N. Ramsey and S. P. Jones. Featherweight concurrency in a portable assembly language, 2000.

[20] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, LNCS, pages 366–381. Springer-Verlag, 2000.

[21] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2001.