Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language

Vasco T. Vasconcelos
LASIGE & DI-FCUL,
University of Lisbon, Portugal
vv@di.fc.ul.pt

Francisco Martins
LASIGE & DI-FCUL,
University of Lisbon, Portugal
fmartins@di.fc.ul.pt

Tiago Cogumbreiro
LASIGE & DI-FCUL,
University of Lisbon, Portugal
cogumbreiro@di.fc.ul.pt

We previously developed a polymorphic type system and a type checker for a multithreaded lock-based polymorphic typed assembly language (MIL) that ensures that well-typed programs do not encounter race conditions. This paper extends such work by taking into consideration deadlocks. The extended type system verifies that locks are acquired in the proper order. Towards this end we require a language with annotations that specify the locking order. Rather than asking the programmer (or the compiler's backend) to specifically annotate each newly introduced lock, we present an algorithm to infer the annotations. The result is a type checker whose input language is non-decorated as before, but that further checks that programs are exempt from deadlocks.

1 Introduction

Type systems for lock-based race and deadlock static detection try to contradict the idea put forward by some authors that "the association between locks and data is established mostly by convention" [15]. Despite all the pathologies usually associated with locks (in the aforementioned article and others), and specially at system's level, locks are here to stay [8].

Deadlock detection should be addressed at the appropriate level of abstraction, for, in general, compiled code that does not deadlock allows us to conclude nothing of the source code. Nevertheless, the problem remains valid at the assembly level and fits quite nicely in the philosophy of typed assembly languages [14]. By capturing a wider set of semantic properties, including the absence of deadlocks, we improve compiler certification in systems where code must be checked for safety before execution, in particular those with untrusted or malicious components.

Our language targets a shared-memory machine featuring an array of processors and a thread pool common to all processors [10, 17]. The thread pool holds threads for which no processor is available, a scheduler chooses a thread from this pool should a processor become idle. Threads voluntary release processors—our model fits in the cooperative multi-threading category. For increased flexibility (and unlike many other models, including [12]) we allow forking threads that hold locks, hence we allow the suspension of processes while in critical regions. A prototype implementation can be found in .

The code in Figure 1 presents a typical example of a potential deadlock comprising a cycle of threads where each thread requests a lock hold by the next thread. Imagine the code running on a two-processors machine: after main completes its execution, each philosopher embarks on a busy-waiting loop, only that two of them will be running in processors, while the third is (and will indefinitely remain) in the run-pool. Situations of deadlocks comprising suspended code are known to be difficult to deal with [13]. Our notion of deadlocked state takes into account running and suspended threads.

Another source of difficulties in characterizing deadlock states derives from the low-level nature of our language that decouples the action of lock acquisition from that of entering a critical section, and

Submitted to:
PLACES 2009

© Vasconcelos, Martins, and Cogumbreiro
This work is licensed under the
Creative Commons Attribution License.

```
main () {
  f1,r3 := newLock; f3,r5 := newLock; f2,r4 := newLock -- 3 forks
  r1:= r3; r2:= r4; fork liftLeftFork[f1,f2] -- 1st philosopher
  r1:= r4; r2:= r5; fork liftLeftFork[f2,f3] -- 2nd philosopher
  r1:= r5; r2:= r3; fork liftLeftFork[f3,f1] -- 3rd philosopher
  done
}
liftLeftFork ∀[l,m].(r1:⟨l⟩^l, r2:⟨m⟩^m) {
  r3:= testSetLock r1
  if r3= 0 jump liftRightFork[l,m]
  jump liftLeftFork[l,m]
}
liftRightFork ∀[l,m].(r1:⟨l⟩^l, r2:⟨m⟩^m) requires {l} {
  r3:= testSetLock r2
  if r3= 0 jump eat[l,m]
  jump liftRightFork[l,m]
}
eat ∀[l,m].(r1:⟨l⟩^l, r2:⟨m⟩^m) requires {l,m} {
  -- eat
  unlock r1-- lay down the left fork
  unlock r2-- lay down the right fork
  -- think
  jump liftLeftFork[l,m]
}
```

Figure 1: The dining philosophers written in MIL

that features non-blocking instructions only. As such the meaning of “entering a critical section” cannot be of a syntactic nature.

A characteristic of our machine is the syntactic dissociation of the test-and-set-lock and the jump-to-critical operations, for which we provide two distinct instructions, as found in conventional instruction sets. Furthermore, there is no syntactic distinction between a conventional conditional jump and a (conditional) jump-to-critical instruction, and the test-set-lock and jump-to-critical instructions can be separated by arbitrary assembly code. As far as the type system goes, the thread holds the lock only after the conditional jump, even though at runtime it may have been obtained long before.

The main contribuitons of this paper are:

- A type system for deadlock elimination. We devise a type system that establishes a strict partial order on lock acquisition, hence enforcing that well typed MIL programs do not deadlock—Theorem 4;
- An algorithm for automatic program annotation. In order to check the absence of deadlock, MIL programs must be annotated to reflect the order by which locks must be acquired. Annotating large assembly programs, either manually or as the result of a compilation process, is not plausible. We present an algorithm that takes a plain MIL program and produces an annotated program together with a collection of constraints over lock sets that are passed to a constraint solver. In case the constraints are solvable the annotated program is typeble—Theorem 8—hence free from deadlocks.

registers	$r ::=$	$\mathsf{r}_1 \mid \dots \mid \mathsf{r}_\mathsf{R}$
lock values	$b ::=$	$\mathbf{0} \mid \mathbf{1} \mid \mathbf{0}^\lambda$
values	$v ::=$	$r \mid n \mid b \mid l \mid v[\lambda] \mid ?\tau$
instructions	$\iota ::=$	
control flow		$r := v \mid r := r + v \mid \mathsf{if}\ r = v\ \mathsf{jump}\ v \mid \mathsf{fork}\ v$
memory		$r := \mathsf{malloc}\ [\vec{\tau}]^\lambda \mid r := v[n] \mid r[n] := v \mid$
locking		$\lambda\colon(\Lambda,\Lambda), r := \mathsf{newLock} \mid r := \mathsf{testSetLock}\ v \mid \mathsf{unlock}\ v$
inst. sequences	$I ::=$	$\iota; I \mid \mathsf{jump}\ v \mid \mathsf{done}$
types	$\tau ::=$	$\mathsf{int} \mid \lambda \mid \langle\vec{\tau}\rangle^\lambda \mid \Gamma\ \mathsf{requires}\ \Lambda \mid \forall[\lambda\colon(\Lambda,\Lambda)].\tau$
register file types	$\Gamma ::=$	$\mathsf{r}_1\colon \tau_1, \dots, \mathsf{r}_n\colon \tau_n$
permissions	$\Lambda ::=$	$\lambda_1, \dots, \lambda_n$
heaps	$H ::=$	$\{l_1\colon h_1, \dots, l_n\colon h_n\}$
heap values	$h ::=$	$\langle v_1 \dots v_n\rangle^\lambda \mid \tau\{I\}$
thread pool	$T ::=$	$\{\langle l_1[\vec{\lambda}_1], R_1\rangle, \dots, \langle l_n[\vec{\lambda}_n], R_n\rangle\}$
register files	$R ::=$	$\{\mathsf{r}_1\colon v_1, \dots, \mathsf{r}_\mathsf{R}\colon v_\mathsf{R}\}$
processors array	$P ::=$	$\{1\colon p_1, \dots, \mathsf{N}\colon p_\mathsf{N}\}$
processor	$p ::=$	$\langle R; \Lambda; I\rangle$
states	$S ::=$	$\langle H; T; P\rangle \mid \mathsf{halt}$

Figure 2: Syntax.

The outline of this paper is as follows. The next section introduces the syntax of programs and machine states, together with the running example. Then Section 3 presents the operational semantics and the notion of deadlocked states. Section 4 describes the type system and the first main result, typable states do not deadlock. Section 5 introduces the annotation algorithm and the second main result, the correctness of the algorithm with respect to the type system. Finally, Section 6 describes related work and concludes the paper.

2 Syntax

The syntax of our language is generated by the grammar in Figure 2. We rely on two mutually disjoint sets for *heap labels*, ranged over by l, and for *singleton lock types*, ranged over by λ. Letter n ranges over integer values.

Values v comprise registers r, integer values n, lock values b, labels l, type application $v[\lambda]$, and uninitialised values $?\tau$. Lock value $\mathbf{0}$ represents an open lock, whereas lock value $\mathbf{1}$ denotes a closed lock; the λ annotation in $\mathbf{0}^\lambda$ allows to determine the lock guarding the critical section a processor is trying to enter and will be useful when defining deadlocked states. Lock values are runtime entities, they need to be distinct from conventional integer values for typing purposes only. Labels are used as heap addresses. Uninitialised values represent meaningless data of a certain type.

Most of the machine instructions ι presented in Figure 2 are standard in assembly languages. Distinct in MIL are the instructions for creating new threads—fork places in the run queue a new thread waiting for execution—, for allocating memory—$\mathsf{malloc}[\tau_1,\dots,\tau_n]^\lambda$ allocates a tuple in the heap protected by lock λ and comprising n cells each of which containing an uninitialised value of type τ_i—, and for manipulating locks. In this last group one finds newLock to create a lock in the heap and store its address in register r (λ describes the singleton lock type associated to the new lock, further described below), testSetLock to acquire a lock, and unlock to release a lock.

Instructions are organised in sequences I, ending in jump or in done. Instruction done terminates a thread, voluntarily releasing the core, giving rise to a cooperative multi-threading model of computation.

Types τ include the integer type int, the singleton lock type λ, the tuple type $\langle\vec{\tau}\rangle^\lambda$ describing a tuple in the heap protected by lock λ, and the code type $\forall[\vec{\lambda}:(\vec{\Lambda},\vec{\Lambda})].(\Gamma\ \mathsf{requires}\ \Lambda)$ representing a code block abstracted on singleton lock types $\vec{\lambda}$, expecting registers of the types in Γ and requiring locks as in Λ. Each universal variable is bound by two sets of singleton lock types Λ, used for deadlock prevention, as described below. For simplicity we allow polymorphism over singleton lock types only; for abstraction over arbitrary types see [10].

The *abstract machine* is parametric on the number of available processors N, and on the number of registers per processor R. An abstract machine can be in two possible states S: halted or running. A running machine comprises a heap H, a thread pool T, and an array of processors P of fixed length N. Heaps are maps from labels l into *heap values* h that may be either data tuples or code blocks. *Tuples* $\langle v_1,\dots,v_n\rangle^\lambda$ are vectors of mutable values v_i protected by some lock λ. Code blocks $\forall[\vec{\lambda}:(\vec{\Lambda},\vec{\Lambda})].(\Gamma\ \mathsf{requires}\ \Lambda)\{I\}$ comprise a signature (a code type) and an instruction sequence I, to be executed by a processor. A thread pool T is a multiset of pairs $\langle l[\vec{\lambda}],R\rangle$, each of which contains the address (a label) of a code block in the heap, a sequence of singleton lock types to act as arguments to the forall type of the code block, and a register file. A processor array P contains N processors, each of which is composed of a register file R mapping the processor's registers to values, a set of locks Λ (the locks held by the thread running at the processor, often call the thread's *permission*), and a sequence of instructions I (the instructions that remain to execute).

Lock order annotations Deadlocks are usually prevented by imposing a strict partial order on locks, and by respecting this order when acquiring locks [4, 9, 12]. The syntax in Figure 2 introduces annotations that specify the locking order. When creating a new lock, we declare the order between the newly introduced singleton lock type and the locks known to the program. We use the notation $\lambda:(\Lambda_1,\Lambda_2)$ to mean that lock type λ is greater than all lock types in set Λ_1 and smaller than each lock type in set Λ_2. The annotated syntax differs from the original syntax ([10, 17]) in two places: at lock creation $\lambda:(\Lambda,\Lambda), r := \mathsf{newLock}$; and in universal types $\forall[\lambda:(\Lambda,\Lambda)].\tau$, where we explicitly specify the lock order on newly introduced singleton lock types.

Example Figure 1 shows an example of a *non-annotated* program. Annotating such a program requires describing the order for each lock introduced in code block main, say,

```
f1::({},{}), r3:= newLock
f3::({f1},{}), r5:= newLock
f2::({f1},{f3}), r4:= newLock
```

and at the types for the three code blocks below.

liftLeftFork $\forall$[l::({},{})].$\forall$ [m::({l},{})].(r1:$\langle$l$\rangle^{\mathsf{l}}$, r2:$\langle$m$\rangle^{\mathsf{m}}$)

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

$$\frac{P(i) = \langle \_\,;\_\,;\mathsf{done}\rangle \qquad H(l) = \forall[\vec{\lambda} : (\_,\_)].(\_\ \mathsf{requires}\ \Lambda)\{I\}}{\langle H;T \uplus \{\langle l[\vec{\lambda}'],R\rangle\};P\rangle \rightarrow \langle H;T;P\{i\colon \langle R;\Lambda;I\rangle[\vec{\lambda}'/\vec{\lambda}]\}\rangle} \qquad \text{(R-SCHEDULE)}$$

$$\frac{P(i) = \langle R;\Lambda \uplus \Lambda';(\mathsf{fork}\ v;I)\rangle \qquad \hat{\mathrm{R}}(v) = l[\vec{\lambda}] \qquad H(l) = \forall[\_].(\_\ \mathsf{requires}\ \Lambda')\{\_\}}{\langle H;T;P\rangle \rightarrow \langle H;T \cup \{\langle l[\vec{\lambda}],R\rangle\};P\{i\colon \langle R;\Lambda;I\rangle\}\rangle} \qquad \text{(R-FORK)}$$

$$\frac{P(i) = \langle R;\Lambda;(\lambda : (\_,\_), r := \mathsf{newLock};I)\rangle \quad l \notin \mathrm{dom}(H) \quad \lambda' \text{ fresh}}{\langle H;T;P\rangle \rightarrow \langle H\{l\colon \langle \mathbf{0}\rangle^{\lambda'}\};T;P\{i\colon \langle R\{r\colon l\};\Lambda;I[\lambda'/\lambda]\rangle\}\rangle} \qquad \text{(R-NEWLOCK)}$$

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

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

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

Figure 3: Operational semantics (thread pool and locks).

```
liftRightFork ∀[l::({},{})].∀[m::({l},{})].(r1:⟨l⟩^l, r2:⟨m⟩^m) requires {l}
eat ∀[l::({},{})].∀[m::({l},{})].(r1:⟨l⟩^l, r2:⟨m⟩^m) requires {l,m}
```

Notice that abstracting one lock at a time, as in the types just shown, precludes declaring code blocks with non-strict partial orders on locks, such as $\forall[l\colon (\emptyset,\{m\}), m\colon (\{l\},\emptyset)].\tau$, which cannot be fulfilled by any conceivable sequence of instructions.

3 Operational Semantics and Deadlocked States

The operational semantics is defined in Figures 3 and 4. The scheduling model of our machine is described by the first three rules in Figure 3. The machine halts when all processors are idle and the thread pool is empty (rule R-HALT). An idle processor (a processor that executes instruction done) picks up an arbitrary thread from the thread pool and activates it (rule R-SCHEDULE); the argument locks $\vec{\lambda}'$ replace the parameters $\vec{\lambda}$ in the code for the processor. For a fork instruction, the machine creates a "closure" by putting together the code label plus its arguments, $l[\vec{\lambda}]$, and a copy of the registers, R, and by placing it in the thread pool. The thread permission is partitioned in two: one part (Λ) stays with the thread, the other (Λ') goes with the newly created thread, as required by the type of its code.

Some rules rely on the evaluation function $\hat{\mathrm{R}}$ that looks for values in registers and in value application.

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

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

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

$$\frac{P(i) = \langle R;\Lambda;(r[n] := v;I)\rangle \qquad R(r) = l \qquad H(l) = \langle v_1..v_n..v_{n+m}\rangle^{\lambda} \qquad \lambda \in \Lambda}{\langle H;T;P\rangle \rightarrow \langle H\{l\colon \langle v_1..\hat{\mathrm{R}}(v)..v_{n+m}\rangle^{\lambda}\};T;P\{i\colon \langle R;\Lambda;I\rangle\}\rangle} \quad \text{(R-STORE)}$$

$$\frac{P(i) = \langle R;\Lambda;\mathsf{jump}\ v\rangle \qquad \hat{\mathrm{R}}(v) = l[\vec{\lambda}] \qquad H(l) = \forall[\vec{\lambda}'\colon (\_,\_)].\_\{I\}}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R;\Lambda;I[\vec{\lambda}/\vec{\lambda}']\rangle\}\rangle} \quad \text{(R-JUMP)}$$

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

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

$$\frac{P(i) = \langle R;\Lambda;(\mathsf{if}\ r = v\ \mathsf{jump}\ v';\_)\rangle \qquad R(r) = v \qquad \hat{\mathrm{R}}(v') = l[\vec{\lambda}] \qquad H(l) = \forall[\vec{\lambda}'\colon (\_,\_)].\_\{I\}}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R;\Lambda;I[\vec{\lambda}/\vec{\lambda}']\rangle\}\rangle} \quad \text{(R-BRANCHT)}$$

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

Figure 4: Operational semantics (memory and control flow).

In our model the heap tuple $\langle \mathbf{0}\rangle^{\lambda}$ represents an open lock, whereas $\langle \mathbf{1}\rangle^{\lambda}$ represents a closed lock. A lock is an uni-dimensional tuple holding a *lock value* because the machine provides for tuple allocation only; lock λ is used for type safety purposes, just like all other singleton lock types. Instruction newLock creates a new open lock in the heap and places a reference l to it in register r. Instruction testSetLock loads the contents of the lock tuple into register r and sets the heap value to $\langle \mathbf{1}\rangle^{\lambda}$; it also makes sure that the lock is not in the thread's permission (rules R-TSL0 and R-TSL1). Further, applying the instruction to an unlocked lock adds lock λ to the permission of the processor (rule R-TSL0). Locks are waved using instruction unlock, as long as the thread holds the lock (rule R-UNLOCK).

Rules related to memory manipulation are described in Figure 4. Rule R-MALLOC creates an heap-allocated λ-protected uninitialised tuple and moves its address to register r. To store values in, and load from, a tuple we require that the lock that guards the tuple is among the processor's permission. In rules R-BRANCHT and R-BRANCHF, we ignore the lock annotation on lock values, so that $\mathbf{0}^{\lambda}$ is considered equal to $\mathbf{0}$. The remaining rules are standard (cf [14]).

Deadlocked States The difficulty in characterising deadlock states stems from the fact that processors never block and that threads may become (voluntary) suspended while in critical a region. We aim at capturing conventional techniques for acquiring locks, namely busy-waiting and sleep-lock [17]. Towards this end, we need to restrict reduction of a given state S to that of a single processor in order to

$$\frac{\Psi \vdash \lambda\colon (\Lambda_1, \_) \qquad \lambda_1 \in \Lambda_1}{\Psi \vdash \lambda_1 \prec \lambda} \qquad \frac{\Psi \vdash \lambda\colon (\_, \Lambda_2) \qquad \lambda_2 \in \Lambda_2}{\Psi \vdash \lambda \prec \lambda_2} \qquad \frac{\Psi \vdash \lambda_1 \prec \lambda_2 \qquad \Psi \vdash \lambda_2 \prec \lambda_3}{\Psi \vdash \lambda_1 \prec \lambda_3}$$

$$\frac{\Psi \vdash \lambda_1 \prec \lambda \quad \cdots \quad \Psi \vdash \lambda_n \prec \lambda}{\Psi \vdash \{\lambda_1, \dots, \lambda_n\} \prec \lambda} \qquad \frac{\Psi \vdash \lambda \prec \lambda_1 \quad \cdots \quad \Psi \vdash \lambda \prec \lambda_n}{\Psi \vdash \lambda \prec \{\lambda_1, \dots, \lambda_n\}}$$

Figure 5: Less-than relation on locks and permissions

$$\frac{\mathrm{ftv}(\tau) \subseteq \mathrm{dom}(\Psi)}{\Psi \vdash \tau} \qquad \frac{\Psi \vdash \tau_i}{\Psi \vdash \mathsf{r}_1\colon \tau_1, \dots, \mathsf{r}_{n+m}\colon \tau_{n+m} <: \mathsf{r}_1\colon \tau_1, \dots, \mathsf{r}_n\colon \tau_n} \qquad \text{(T-TYPE,S-REGFILE)}$$

$$\frac{\Psi \vdash \tau}{\Psi, l\colon \tau; \Gamma \vdash l\colon \tau} \qquad \frac{\Psi \vdash \tau}{\Psi; \Gamma_1, r_i\colon \tau, \Gamma_2 \vdash r_i\colon \tau} \qquad \Psi;\Gamma \vdash n\colon \mathsf{int} \qquad \Psi;\Gamma \vdash \mathbf{0}, \mathbf{1}, \mathbf{0}^{\lambda}\colon \lambda \qquad \Psi;\Gamma \vdash ?\tau\colon \tau$$

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

$$\frac{\Psi \vdash \lambda' \qquad \Psi;\Gamma \vdash v\colon \forall[\lambda\colon (\Lambda_1, \Lambda_2)]\tau \qquad \Psi \vdash \Lambda_1 \prec \lambda' \prec \Lambda_2}{\Psi;\Gamma \vdash v[\lambda']\colon \tau[\lambda'/\lambda]} \qquad \text{(T-VALAPP)}$$

Figure 6: Rules for values $\boxed{\Psi;\Gamma \vdash v\colon \tau}$, for subtyping $\boxed{\Psi \vdash \Gamma <: \Gamma}$, and for types $\boxed{\Psi \vdash \tau}$.

control the progress of a single core: let relation $S \rightarrow_i S'$ denote a reduction step on processor i excluding rules R-HALT, R-SCHEDULE and R-UNLOCK.

Definition 1 (Deadlocked states)**.** *Let S be the state $\langle H;T;P\rangle$.*

- *A processor $\langle R;\Lambda;I\rangle$ holds lock λ when $\lambda \in \Lambda$; a suspend thread $\langle l[\vec{\lambda}'], R\rangle$ holds lock λ when $H(l) = \forall[\vec{\lambda}\colon (\_,\_)].(\_\ \mathsf{requires}\ \Lambda)\{\_\}$ and $\lambda \in \Lambda[\vec{\lambda}'/\vec{\lambda}]$;*
- *A* processor p in P immediately tries to enter a critical section guarded by lock λ *if p is of the form $\langle R;\_;(\mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v;\_)\rangle$ and $R(r) = \mathbf{0}^{\lambda}$;*
- *For busy waiting, a* thread in processor p_i is trying to enter a critical region guarded by λ *if $S \rightarrow_i^* S'$ and processor p_i in state S' immediately tries to enter a critical section guarded by λ;*
- *For sleep-lock, a* thread $\langle l[\vec{\lambda}'], R\rangle$ in thread pool T is trying to enter a critical region guarded by λ *if $H(l) = \forall[\vec{\lambda}\colon \_].(\_\ \mathsf{requires}\ \Lambda)\{I\}$, and the thread in processor p_1 of state $S + P\{1\colon \langle R;\Lambda;I\rangle[\vec{\lambda}'/\vec{\lambda}]\}$ is trying to enter a critical region guarded by λ;*
- *A state S is* deadlocked *if there exist locks $\lambda_0, \dots, \lambda_n$, with $\lambda_0 = \lambda_n$, and indices $d_0, \dots, d_{n-1}$ ($n > 0$) such that for each $0 \leq i < n$, either processor p_{d_i} or suspended thread t_{d_i} holds lock λ_i and is trying to enter a critical region guarded by λ_{i+1}.*

Notice that $d_i \neq d_j$ does not imply $p_{d_i} \neq p_{d_j}$ and similarly for threads in the thread pool, so that a state deadlocked on locks $\lambda_0, \dots, \lambda_n$ may involve less than n threads. We have excluded the R-UNLOCK rule from the $\rightarrow_i$ reduction relation, yet releasing a lock is not necessarily an indication that the thread is leaving a deadlocked state, for the released lock may not be involved in the deadlock; a more general definition of deadlocked state would take this fact into account.

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

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

$$\frac{\Psi, \lambda\colon (\Lambda_1,\Lambda_2);\Gamma\{r\colon \langle\lambda\rangle^{\lambda}\};\Lambda \vdash I \qquad \lambda \notin \Psi,\Gamma,\Lambda}{\Psi;\Gamma;\Lambda \vdash \lambda\colon (\Lambda_1,\Lambda_2), r := \mathsf{newLock}; I} \qquad \text{(T-NEWLOCK)}$$

$$\frac{\Psi;\Gamma \vdash v\colon \langle\lambda\rangle^{\lambda} \qquad \Psi;\Gamma\{r\colon \lambda\};\Lambda \vdash I \qquad \lambda \notin \Lambda}{\Psi;\Gamma;\Lambda \vdash r := \mathsf{testSetLock}\ v; I} \qquad \text{(T-TSL)}$$

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

$$\frac{\Psi;\Gamma \vdash r\colon \lambda \qquad \Psi;\Gamma \vdash v\colon \Gamma' \ \mathsf{requires}\ \Lambda \uplus \{\alpha\} \qquad \Psi;\Gamma;\Lambda \vdash I \qquad \Psi \vdash \Gamma <: \Gamma' \qquad \Psi \vdash \Lambda \prec \lambda}{\Psi;\Gamma;\Lambda \vdash \mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I} \qquad \text{(T-CRITICAL)}$$

Figure 7: Typing rules for instructions (thread pool and locks) $\boxed{\Psi;\Gamma;\Lambda \vdash I}$.

4 A Type System for Deadlock Prevention

Type System Typing environments Ψ map heap addresses l to types τ, and singleton lock types λ to lock kinds (Λ_1,Λ_2). An entry $\lambda\colon (\Lambda_1,\Lambda_2)$ in Ψ means that λ is larger than all lock types in Λ_1 and smaller than any lock type in Λ_2, a notion captured by relation $\prec$ described in Figure 5. Instructions are also checked against a register file type Γ holding the current types of the registers, and a set Λ of lock variables: the *permission* of (the processor executing) the code block. The type system is presented in Figures 6 to 9.

Typing rules for values are illustrated in Figure 6. Rule T-TYPE makes sure types are *well-formed*, that all free singleton lock types (or free type variables, ftv) in a type are bound in the typing environment. A formula $\Gamma <: \Gamma'$ allows "forgetting" registers in the register file type, and is particularly useful in jump instructions where we want the type of the target code block to be more general (ask for less registers) than those active in the current code [14]. The rule for value application, T-VALAPP, checks that the argument λ' is within the interval (Λ_1,Λ_2), as required by the parameter λ.

The rules in Figure 7 capture the policy for lock usage. Rule T-DONE requires the release of all locks before terminating the thread. Rule T-FORK splits permissions into sets Λ and Λ': the former is transferred to the forked thread according to the permissions required by the target code block, the latter remains with the processor. Rule T-NEWLOCK assigns a lock type $\langle\lambda\rangle^{\lambda}$ to the register. The new singleton lock type λ is recorded in Ψ, so that it may be used in the rest of the instructions I. Rule T-TSL requires that the value under test is a lock in the heap (of type $\langle\lambda\rangle^{\lambda}$) and records the type of the lock value λ in register r. This rule also disallows testing a lock already held by the processor. Rule T-UNLOCK makes sure that only held locks are unlocked. Rule T-CRITICAL ensures that the processor holds the permission required by the target code block, including the lock under test. A processor is guaranteed to hold the tested lock only after (conditionally) jumping to the critical region. A previous test-and-set-lock instructions may have obtained the lock, but the type system records that the processor

$$\frac{\Psi;\Gamma\{r\colon \langle\vec{\tau}\rangle^{\lambda}\};\Lambda \vdash I \qquad \tau_i \neq \lambda \qquad \lambda \in \Lambda}{\Psi;\Gamma;\Lambda \vdash r := \mathsf{malloc}\ [\vec{\tau}]^{\lambda}; I} \quad (\text{T-MALLOC})$$

$$\frac{\Psi;\Gamma \vdash v\colon \langle\tau_1..\tau_{n+m}\rangle^{\lambda} \qquad \Psi;\Gamma\{r\colon \tau_n\};\Lambda \vdash I \qquad \tau_n \neq \lambda' \qquad \lambda \in \Lambda}{\Psi;\Gamma;\Lambda \vdash r := v[n]; I} \quad (\text{T-LOAD})$$

$$\frac{\Psi;\Gamma \vdash v\colon \tau_n \qquad \Psi;\Gamma \vdash r\colon \langle\tau_1..\tau_{n+m}\rangle^{\lambda} \qquad \Psi;\Gamma\{r\colon \langle\tau_1..\tau_{n+m}\rangle^{\lambda}\};\Lambda \vdash I \qquad \tau_n \neq \lambda' \qquad \lambda \in \Lambda}{\Psi;\Gamma;\Lambda \vdash r[n] := v; I} \quad (\text{T-STORE})$$

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

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

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

$$\frac{\Psi;\Gamma \vdash v\colon \Gamma'\ \mathsf{requires}\ \Lambda \qquad \Psi \vdash \Gamma <: \Gamma'}{\Psi;\Gamma;\Lambda \vdash \mathsf{jump}\ v} \quad (\text{T-JUMP})$$

Figure 8: Typing rules for instructions (memory and control flow) $\boxed{\Psi;\Gamma;\Lambda \vdash I}$.

holds the lock only after the conditional jump. The rule checks that the newly acquired lock is larger than all locks in the possession of the thread.

The typing rules for memory and control flow are depicted in Figure 8. Operations for loading from (T-LOAD), and for storing into (T-STORE), tuples require that the processor holds the right permissions (the locks for the tuples it reads from, or writes to). Both rules preclude the direct manipulation of lock values by programs, via the $\tau_n \neq \lambda'$ assumptions.

The rules for typing machine states are illustrated in Figure 9. The rule for a thread item in the thread pool checks that the type and required registers R are as expected in the type of the code block pointed by v. Similarly, the rule for type checking a processor also permits that type Γ of the registers R be more specific than the register file type Γ' required to type check the remaining instructions I. The heap value rule for code blocks adds to Ψ each singleton lock type (together with its bounds), so that they may be used in the rest of the instructions I.

Example As expected, the example is not typable with the annotations introduced previously. The three **newLock** instructions place in Ψ three entries $f_1\colon (\emptyset,\emptyset)$, $f_2\colon (\{f_1\},\{f_3\})$, $f_3\colon (\{f_1\},\emptyset)$. Then the value (liftLeftFork[f2])[f1] (in the example: liftLeftFork[f1,f2]) in the first **fork** instruction issues goals $\Psi \vdash \emptyset \prec f_1 \prec \emptyset$ and $\Psi \vdash \{f_1\} \prec f_2 \prec \emptyset$, which are easy to guarantee given that Ψ contains an entry $f_2\colon (\{f_1\},\{f_3\})$. Likewise, the second **fork** instruction, generates goals $\Psi \vdash \emptyset \prec f_2 \prec \emptyset$ and $\Psi \vdash \{f_2\} \prec f_3 \prec \emptyset$, which are again hold because of same entry. However, the last **fork** instruction requires $\Psi \vdash \emptyset \prec f_3 \prec \emptyset$ and $\Psi \vdash \{f_3\} \prec f_2 \prec \emptyset$, the second of which does not hold.

Notice however that each of the three **jump** instructions are typable per se. For example, in code block liftRightFork, instruction **if** r$_3$= **0** **jump** eat[l,m] requires $\Psi \vdash \{l\} \prec m$, which holds because the signature for the code block includes the annotation $m\colon (\{l\},\emptyset)$.

$$\frac{\forall i.\Psi \vdash \Gamma(\mathsf{r}_i) \qquad \Psi;\emptyset \vdash R(\mathsf{r}_i) \colon \Gamma(\mathsf{r}_i)}{\Psi \vdash R \colon \Gamma} \qquad \text{(reg file, } \boxed{\Psi \vdash R\colon \Gamma}\text{)}$$

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

$$\frac{\forall i.\Psi \vdash t_i}{\Psi \vdash \{t_1,\dots,t_n\}} \qquad \frac{\Psi;\emptyset \vdash v\colon \Gamma' \text{ requires } \_ \qquad \Psi \vdash R\colon \Gamma \qquad \Psi \vdash \Gamma <: \Gamma'}{\Psi \vdash \langle v,R\rangle} \qquad \text{(thread pool, } \boxed{\Psi \vdash T}\text{)}$$

$$\frac{\tau = \forall[\vec{\lambda}\colon (\vec{\Lambda}_1,\vec{\Lambda}_2)].(\Gamma \text{ requires } \Lambda) \qquad \Psi,\vec{\lambda}\colon (\vec{\Lambda}_1,\vec{\Lambda}_2);\Gamma;\Lambda \vdash I}{\Psi \vdash \tau\{I\}\colon \tau} \qquad \frac{\forall i.\Psi;\emptyset \vdash v_i\colon \tau_i}{\Psi \vdash \langle \vec{v}\rangle^{\lambda} \colon \langle \vec{\tau}\rangle^{\lambda}}$$

$$\text{(heap value, } \boxed{\Psi \vdash h\colon \tau}\text{)}$$

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

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

Figure 9: Typing rules for machine states.

Typable States Do Not Deadlock The main result of the type system, namely that $\Psi \vdash S$ and $S \rightarrow^* S'$ implies S' not deadlocked, follows from Subject Reduction and from Typable States Are Not Deadlocked, in a conventional manner.

Lemma 2 (Substitution Lemma). *If $\Psi,\lambda\colon (\Lambda_1,\Lambda_2);\Gamma,\Lambda \vdash I$ and $\Psi(\lambda)' = (\Lambda_1,\Lambda_2)$, then $\Psi;\Gamma\sigma,\Lambda\sigma \vdash I\sigma$, where $\sigma = [\lambda'/\lambda]$.*

Theorem 3 (Subject Reduction). *If $\Psi \vdash S$ and $S \rightarrow S'$, then $\Psi' \vdash S'$, where $\Psi' = \Psi$ or $\Psi' = \Psi, l\colon \langle\vec{\tau}\rangle^{\lambda}$ (with l fresh) or $\Psi' = \Psi, l\colon \langle\lambda\rangle^{\lambda}, \lambda\colon (\vec{\Lambda}_1,\vec{\Lambda}_2)$ (with l,λ fresh).*

Proof. (Outline) By induction on the derivation of $S \rightarrow S'$ proceeding by case analysis on the last rule of the derivation, using the substitution lemma for rules R-SCHEDULE, R-FORK, R-JUMP, and R-BRANCHT, as well as weakening in several rules. □

Theorem 4 (Typable States Are Not Deadlocked). *If $\Psi \vdash S$, then S is not deadlocked.*

Proof. (Sketch) Consider the contra-positive and show that deadlocked states are not typable. Without loss of generality suppose that S is of the form $\langle H;\langle t_{d_0},\dots,t_{d_m}\rangle;\{d_{m+1}\colon p_{d_{m+1}},\dots,d_n\colon p_{d_n}\}\rangle$ with suspended threads t_{d_i} and processors p_{d_j} not necessarily distinct.

Each of these threads and processors are trying to enter a critical region. For a processor p_{d_i} we have that $S \rightarrow^*_{d_i} S'$ where d_i-th processor in S' is of the form $\langle R;\Lambda;(\mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v;\_)\rangle$ and $R(r) = \langle\_\rangle^{\lambda^{d_{i+1}}}$. By Subject Reduction $\Psi' \vdash S'$ where Ψ' extends Ψ as stated in Theorem 3. A simple derivation starting from rule T-CRITICAL allows to conclude that $\Psi \vdash \Lambda_{d_i} \prec \lambda_{d_{i+1}}$. For a thread $t_{d_i} = \langle l[\vec{\lambda}'],R\rangle$ in thread pool of S we run the machine S'' obtained from S by replacing processor 1 with $\langle R;\Lambda;I\rangle[\vec{\lambda}'/\vec{\lambda}]$, where $H(l) = \forall[\vec{\lambda}\colon \_].(\_ \text{ requires } \Lambda)\{I\}$. Proceeding as for processes above, we conclude again that $\Psi \vdash \Lambda_{d_i} \prec \lambda_{d_{i+1}}$.

We thus have $\Psi \vdash \Lambda_{d_0} \prec \lambda_{d_1}, \dots \Psi \vdash \Lambda_{d_{n-1}} \prec \lambda_{d_n}, \Psi \vdash \Lambda_{d_n} \prec \lambda_{d_0}$ which is not satisfiable. □

5 Type Inference

Annotating lock ordering on large assembly programs may not be an easy task. In our setting, programmers (compilers, more often) produce annotation free programs such as the one in Figure 1, and use an inference algorithm to provide for the missing annotations.

The Algorithm The *annotation-free syntax* is obtained from that in Figure 2, by removing the $:(\Lambda,\Lambda)$ part both in the newLock instruction and in the universal type. Given an annotation-free program H, algorithm $\mathscr{W}$ produces a pair, comprising a typing environment Ψ and an annotated program $H^\star$, such that $\Psi \vdash H^\star$, or else fails. In the former case $H^\star$ is typable, hence does not deadlock (Theorem 3); in the latter case, there is no possible labeling for H.

We depend on a set of *variables over permissions* (sets of locks), ranged over by ν, disjoint from the set of heap labels and from the set singleton lock types introduced in Section 2. Constraints are computed by an intermediate step in our algorithm.

Definition 5 (Constraints and solutions)**.**

- *We consider* constraints *of three distinct forms:* $\Lambda \prec \lambda$*,* $\nu \prec \lambda$*, and* $\lambda \prec \nu$*, and denote by* C *a set of constraints;*
- *A* substitution θ *is a map from permission variables* ν *to permissions* Λ*;*
- *A substitution* θ solves (Ψ, C) *if* $\Psi\theta \vdash x\theta \prec y\theta$ *for all* $x \prec y \in C$*.*

Algorithm $\mathscr{W}$ runs in two phases: the first, $\mathscr{A}$, produces a triple comprising a typing environment Ψ, an annotated program $H^\star$, and a collection of constraints C, all containing variables over permissions Λ. The set of constraints is then passed to a constraint solver, that either produces a substitution θ or fails. In the former case, the output of $\mathscr{W}$ is the pair $(\Psi\theta, H^\star\theta)$; in the latter $\mathscr{W}$ fails. In practice, we do not need to generate $H^\star$ or to perform the substitutions; our compiler accepts H if the produced collection of constraints is solvable, and rejects it otherwise.

Generating constraints Algorithm $\mathscr{A}$, described in Figure 10, visits the program twice. On a first step it builds an initial type environment $\Psi_0 = \{l_i\colon \tau_i^\star\}_{i\in I}$ collecting the types for all code blocks in the given program $\{l_i\colon \tau_i\{I_i\}\}_{i\in I}$, annotating with permission variables (denoted by ν and ρ) the intervals for the locks bound in forall types; on a second visit it generates the constraints and the annotated syntax for the instructions in each code block.

The algorithm for instructions, $\mathscr{I}$, also shown in Figure 10, generates annotations for the singleton lock type introduced in **newLock** instructions, or further constraints in the case of the jump-to-critical instruction. In the case of a fork instruction, the algorithm calls function $\mathscr{V}$ to obtain the required permission Λ' and passes the difference $\Lambda \setminus \Lambda'$ to the function that annotates the continuation I.

The algorithm for values, $\mathscr{V}$, generates constraints in the case of type application. Finally, the algorithm for types annotates the singleton lock types in forall types. In the definition of all algorithms, permission-variables $\nu, \rho, \vec{\nu}_i, \vec{\rho}_i$ are freshly introduced.

Example For the running example, we first rename all bound variables so that the type of code block liftLeftFork mentions l_1 and m_1, that of liftLeftFork mentions l_2 and m_2, and that of eat uses l3 and m3. For example:

liftRightFork $\forall[\mathsf{l}_2].\forall[\mathsf{m}_2].(\mathsf{r}_1{:}\langle \mathsf{l}_2\rangle^{\mathsf{l}_2},\ \mathsf{r}_2{:}\langle \mathsf{m}_2\rangle^{\mathsf{m}_2})$ **requires** $\{\mathsf{l}_2\}$

$$\mathscr{A}(\{l_i\colon \tau_i\{I_i\}\}_{i\in I}) = (\cup_{i\in I}\Psi_i, \{l_i\colon \tau_i^\star\{I_i^\star\}\}_{i\in I}, \cup_{i\in I}C_i)$$
$$\quad\text{where } \tau_i^\star = \forall[\vec{\lambda}_i\colon (\vec{v}_i,\vec{\rho}_i)].(\Gamma_i \text{ requires } \Lambda_i) = \mathscr{T}(\tau_i)$$
$$\quad\text{and } (I_i^\star,\Psi_i,C_i) = \mathscr{I}(I_i, \{l_i\colon \tau_i^\star\}_{i\in I} \cup \{\vec{\lambda}_i\colon (\vec{v}_i,\vec{\rho}_i)\}, \Gamma_i, \Lambda_i)$$
$$\mathscr{I}((\lambda, r := \mathsf{newLock}; I),\Psi,\Gamma,\Lambda) = ((\lambda\colon (\nu,\rho), r := \mathsf{newLock}; I^\star),\Psi',C)$$
$$\quad\text{where } (I^\star,\Psi',C) = \mathscr{I}(I, \Psi \uplus \{\lambda\colon (\nu,\rho)\}, \Gamma\{r\colon \langle\lambda\rangle^\lambda\},\Lambda)$$
$$\mathscr{I}((\mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I),\Psi,\Gamma,\Lambda) = ((\mathsf{if}\ r=\mathbf{0}\ \mathsf{jump}\ v; I^\star),\Psi',C_1\cup C_2\cup\{\Lambda \prec \lambda\})$$
$$\quad\text{where } (\Gamma' \text{ requires } (\Lambda\uplus\{\lambda\}),C_1) = \mathscr{V}(v,\Psi,\Gamma)$$
$$\quad\text{and } (I^\star,\Psi',C_2) = \mathscr{I}(I,\Psi,\Gamma,\Lambda)$$
$$\quad\text{and } \Psi \vdash \Gamma <: \Gamma'$$
$$\quad\text{and } \lambda = \Gamma(r)$$
$$\mathscr{I}((\mathsf{fork}\ v; I),\Psi,\Gamma,\Lambda) = ((\mathsf{fork}\ v; I^\star),\Psi',C_1\cup C_2)$$
$$\quad\text{where } (\Gamma' \text{ requires } \Lambda',C_1) = \mathscr{V}(v,\Psi',\Gamma)$$
$$\quad\text{and } (I^\star,\Psi',C_2) = \mathscr{I}(I,\Psi,\Gamma,\Lambda\setminus\Lambda')$$
$$\quad\text{and } \Psi \vdash \Gamma <: \Gamma'$$
$$\mathscr{V}(v[\lambda],\Psi,\Gamma) = (\tau[\lambda/\lambda'], C\cup\{\nu \prec \lambda \prec \rho\})$$
$$\quad\text{where } (\forall[\lambda'\colon (\nu,\rho)]\tau, C) = \mathscr{V}(v,\Psi,\Gamma)$$
$$\mathscr{T}(\forall[\vec{\lambda}].((\mathsf{r}_1\colon \tau_1,\ldots,\mathsf{r}_n\colon \tau_n) \text{ requires } \Lambda)) = \forall[\vec{\lambda}\colon (\vec{\nu},\vec{\rho})].((\mathsf{r}_1\colon \mathscr{T}(\tau_1),\ldots,\mathsf{r}_n\colon \mathscr{T}(\tau_n)) \text{ requires } \Lambda)$$

Figure 10: The tagging algorithm (selected rules).

Then, algorithm $\mathscr{A}$ creates an initial environment Ψ_0 by generating twelve variables (ρ_1 to ρ_{12}) to annotate the six locks ($\mathsf{l_i}$ and $\mathsf{m_i}$) in the three code blocks that mention locks (liftLeftFork, liftRightFork, and eat). They are $\mathsf{l}_1\colon (\rho_1,\rho_2)$, ..., $\mathsf{m}_3\colon (\rho_{11},\rho_{12})$. Revisiting the signature of code block liftRightFork, we get:

liftRightFork $\forall[\mathsf{m}_2{::}(\rho_7,\rho_8)].\forall[\mathsf{l}_2{::}(\rho_5,\rho_6)].(\mathsf{r}_1{:}\langle \mathsf{l}_2\rangle^{\mathsf{l}_2},\ \mathsf{r}_2{:}\langle \mathsf{m}_2\rangle^{\mathsf{m}_2})$ **requires** $\{\mathsf{l}_2\}$

In the second pass, while in code block main, algorithm $\mathscr{I}$ generates six more permission variables (ρ_{13} to ρ_{18}) to annotate the new lock variables f_1 to f_3 introduced with the **newLock** instructions. They are: $\mathsf{f}_1\colon (\rho_{13},\rho_{14})\ldots \mathsf{f}_3\colon (\rho_{17},\rho_{18})$. The rest of the second pass generates new constraints in type application and in jump-to-critical instructions. For example, in code block liftRightFork, and for value eat[l_2,m_2], four constraints are generated: $\rho_9 \prec \mathsf{l}_2 \prec \rho_{10}, \rho_{11} \prec \mathsf{m}_2 \prec \rho_{12}$. Then, in the jump-to-critical instruction, **if** r_3= 0 **jump** eat[l_2,m_2], and since the thread holds lock l_2 (as witnessed by its signature **requires** (l_2)), a new constraint $\{\mathsf{l}_2\} \prec \mathsf{m}_2$ is generated. The thus created set of constraints is then passed to a constraint solver, which is bound to fail.

Main result For soundness we start with a few lemmas.

Lemma 6 (Value soundness). *If $\mathscr{V}(v,\Psi,\Gamma) = (\tau,C)$ and θ solves (Ψ,C) then $\Psi\theta;\Gamma\theta \vdash v\colon \tau\theta$.*

Proof. (Outline) The proof proceeds by induction on the inference tree for $\Psi\theta;\Gamma\theta \vdash v\colon \tau\theta$ performing case analysis on the last typing rule applied. □

Lemma 7 (Instruction soundness). *If* $\mathscr{I}(I,\Psi,\Gamma,\Lambda) = (I^\star,\Psi',C)$ *and* θ *solves* (Ψ',C) *then* $\Psi'\theta;\Gamma\theta;\Lambda\theta \vdash I^\star\theta$ *and* $\Psi \subseteq \Psi'$.

Proof. (Outline) The proof proceeds by induction on I. The cases for conditional jump and fork use Lemma 6. □

Theorem 8 (Soundness). *If* $\mathscr{W}(H) = (\Psi,H^\star)$ *then* $\Psi \vdash H^\star$.

Proof. (Outline) Follows directly from Lemma 7 using typing rules for heap values and heaps. We use weakening on typing environments before applying the heap rule. □

Conversely, we believe that if $\Psi \vdash H^\star$, then $\mathscr{W}(\mathscr{E}(H^\star))$ does not fail, where $\mathscr{E}$ is the obvious lock-order annotation erasure function. A stronger result would include a notion of principal solutions.

6 Related Work and Conclusion

Related work The literature on type systems for deadlock freedom in lock-based languages is vast; space restrictions prohibit a general survey. We however believe that the problem of type inference for deadlock freedom in lock-based languages has been given not so much attention in high-level languages, let alone low-level (assembly) languages. Three characteristics separate our work from most proposals on the topic: the non block structure of the locking primitives, the facts that threads never block and that they may be suspended while holding locks.

Following Coffman *et al.* one can classify the problem of deadlock under the categories of *detection and recovery*, *avoidance* and *prevention* [5, 9]. In the first category, detection and recovery, on finds for example works that check deadlocks at runtime. Cunningham *et al.* infer locks for atomicity in an object-oriented language, but use a runtime mechanism to detect when a thread's lock acquisition would cause a deadlock [11]. Java PathFinder [7] and Driver Verifier [3] identify violations of the lock discipline during runtime tests. Agarwal *et al.* [1, 2] present an algorithm that detects potential deadlocks involving any number of threads.

Under the *avoidance* category on finds, e.g., a recent work by Boudol where a type and effect system allows for the design of an operational semantics that refuses to lock a pointer whenever it anticipates to take a pointer that is held by another thread [5].

Our work falls into the third category above, *prevention*. Flanagan and Abadi present a functional language with mutable references where locking is block structured and threads physically block [12]. From this work we borrowed the idea of singleton lock types to describe, at the type level, a single lock. Type based deadlock prevention has also been study in the realm of object-oriented languages, where, e.g., Boyapati *et al.* use a variant of ownership types for preventing deadlocks in Java, performing partial inference of annotations, but not of those related to lock order [6].

Suenaga proposes a concurrent functional language similar to Flanagan and Abadi's mentioned above, except that it features non block structured locking [16]; his language includes separate primitives for locking/unlocking, as in our case. Albeit targeting at different level of abstraction, the results (deadlock prevention) and the techniques (type inference) in both works are similar, in particular the usage of a constraint-based algorithm to infer types. Differently from our case, Suenaga uses ownership types rather than singleton lock types.

Concluding remarks We have presented a type system that enforces a strict partial order on lock acquisition, guaranteeing that well typed programs do not deadlock. Towards this end we extended the syntax of our language to incorporate annotations on the locking order. Acknowledging that the annotation of large assembly programs (either manually or as the result of a compilation process) is not plausible, we have introduced an algorithm that infers the required annotations. The algorithm is proved to be correct, hence that programs that pass our compiler are exempt from deadlocks.

The current implementation of the algorithm generates, from a non-annotated program, a set of constraints in the form of a Prolog goal. The goal is then checked against a Prolog program that implements the $\prec$ relation in Figure 5. We consider the program typable if the goal succeeds. There is no point in building the annotated syntax or performing the substitution, as explained in Section 5. Future work in this area includes the automation of the whole process either by calling the Prolog interpreter from within the compiler, or by implementing relation $\prec$ directly in Java, the language of our type checker/interpreter.

Future work also includes trying to assess the usage of our type checker on larger programs, generated for example from an imperative high-level language, and to further compare the singleton lock types and the ownership types approaches for the description of non block structured locking.

References

[1] Rahul Agarwal & Scott D. Stoller (2006): *Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables*. In: *Proceedings of PADTAD'06*. ACM, New York, NY, USA, pp. 51–60.

[2] Rahul Agarwal, Liqiang Wang & Scott D. Stoller (2005): *Detecting potential deadlocks with static analysis and runtime monitoring*. In: *Proceedings of PADTAD'05*. Springer, pp. 191–207.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani & Abdullah Ustuner (2006): *Thorough static analysis of device drivers*. *SIGOPS Operating Systems Review* 40(4), pp. 73–85.

[4] Andrew Birrell (1989): *An Introduction to Programming with Threads*. Technical Report 35, Digital Systems Research Center.

[5] Gérard Boudol (2009): *A Deadlock-Free Semantics for Shared Memory Concurrency*. In: Martin Leucker & Carroll Morgan, editors: *Proceedings of ICTAC'09*, *LNCS* 5684. Springer, pp. 140–154.

[6] Chandrasekhar Boyapati, Robert Lee & Martin Rinard (2002): *Ownership types for safe programming: preventing data races and deadlocks*. In: *Proceedings of OOPSLA'02*. ACM, pp. 211–230.

[7] Guillaume Brat, Klaus Havelund, SeungJoon Park & Willem Visser (2000): *Java PathFinder — Second Generation of a Java Model Checker*. In: *Proceedings of the Workshop on Advances in Verification*.

[8] Bryan Cantrill & Jeff Bonwick (2008): *Real-world Concurrency*. *Queue* 6(5), pp. 16–25.

[9] E. G. Coffman, M. Elphick & A. Shoshani (1971): *System Deadlocks*. *ACM Comput. Surv.* 3(2), pp. 67–78.

[10] Tiago Cogumbreiro, Francisco Martins & Vasco T. Vasconcelos (2009): *Compiling the π-calculus into a multithreaded Typed Assembly Language*. *ENTCS* 241, pp. 57–84. Proceedings of PLACES'08.

[11] David Cunningham, Sophia Drossopoulou & Susan Eisenbach (2008): *Lock Inference Proven Correct*. In: *Proceedings of FTfJP'08*.

[12] Cormac Flanagan & Martín Abadi (1999): *Types for Safe Locking*. In: S. Doaitse Swierstra, editor: *Proceedings of ESOP'99*, *LNCS* 1576. Springer, pp. 91–108.

[13] Leonidas I. Kontothanassis, Robert W. Wisniewski & Michael L. Scott (1994): *Scheduler-Conscious Synchronization*. Technical Report, University of Rochester, Rochester, NY, USA.

[14] Greg Morrisett, David Walker, Karl Crary & Neal Glew (1999): *From System F to Typed Assembly Language*. *ACM Transactions on Programing Language and Systems* 21(3), pp. 527–568.

[15] Nir Shavit (2008): *Technical perspective Transactions are tomorrow's loads and stores*. *Communications of the ACM* 51(8), pp. 90–90.

[16] Kohei Suenaga (2008): *Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References*. In: Ganesan Ramalingam, editor: *Proceedings of APLAS'08*, *LNCS* 5356. Springer, pp. 155–170.

[17] Vasco T. Vasconcelos & Francisco Martins (2006): *A Multithreaded Typed Assembly Language*. In: Ganesh Gopalakrishnan & John O'Leary, editors: *Proceedings of TV'06*. pp. 133–141.