Types for Safe Locking

Cormac Flanagan and Martín Abadi
[flanagan|ma]@pa.dec.com

Systems Research Center, Compaq

Abstract. A race condition is a situation where two threads manipulate a data structure simultaneously, without synchronization. Race conditions are common errors in multithreaded programming. They often lead to unintended nondeterminism and wrong results. Moreover, they are notoriously hard to diagnose, and attempts to eliminate them can introduce deadlocks. In practice, race conditions and deadlocks are often avoided through prudent programming discipline: protecting each shared data structure with a lock and imposing a partial order on lock acquisitions. In this paper we show that this discipline can be captured (if not completely, to a significant extent) through a set of static rules. We present these rules as a type system for a concurrent, imperative language. Although weaker than a full-blown program-verification calculus, the type system is effective and easy to apply. We emphasize a core, first-order type system focused on race conditions; we also consider extensions with polymorphism, existential types, and a partial order on lock types.

1 Races, Locks, and Types

Programming with multiple threads introduces a number of pitfalls, such as race conditions and deadlocks. A race condition is a situation where two threads manipulate a data structure simultaneously, without synchronization. Race conditions are common, insidious errors in multithreaded programming. They often lead to unintended nondeterminism and wrong results. Moreover, since race conditions are timing-dependent, they are notoriously hard to track down. Attempts to eliminate race conditions by using lock-based synchronization can introduce other errors, in particular deadlocks. A deadlock occurs when no thread can make progress because each is blocked on a lock held by some other thread.

In practice, both race conditions and deadlocks are often avoided through careful programming discipline [5]. Race conditions are avoided by protecting each shared data structure with a lock, and accessing the data structure only when the protecting lock is held. Deadlocks are avoided by imposing a strict partial order on locks and ensuring that each thread acquires locks only in increasing order. However, this programming discipline is not well supported by existing development tools. It is difficult to check if a program adheres to this discipline, and easy to write a program that does not by mistake. A single unprotected access in an otherwise correct program can produce a timing-dependent race condition whose cause may take weeks to identify.

In this paper we show that this programming discipline can be captured through a set of static rules. We present those rules as a type system for a concurrent, imperative language. We initially consider a first-order type system focused on race conditions. The type system supports dynamic thread creation and the dynamic allocation of locks and reference cells. We then consider extensions, such as universal and existential types, that increase the expressiveness of the system. We also outline an extension that eliminates deadlock by enforcing a strict partial order on lock acquisitions.

Since the programming discipline dictates that a thread can access a shared data structure only when holding a corresponding lock, our type systems provide rules for proving that a thread holds a given lock at a given program point. The rules rely on *singleton lock types*. A singleton lock type is the type of a single lock. Therefore, we can represent a lock l at the type level with the singleton lock type that contains l, and we can assert that a thread holds l by referring to that type rather than to the lock l. The type of a reference cell mentions both the type of the contents of the cell and the singleton lock type of the lock that protects the cell. Thus, singleton lock types provide a simple way of injecting lock values into the type level.

A set of singleton lock types forms a *permission*. During typechecking, each expression is analyzed in the context of a permission; including a singleton lock type in the permission amounts to assuming that the corresponding lock is held during evaluation of the expression. In addition, a permission decorates each function type and each function definition, representing the set of locks that must be held before a function call.

We study typechecking rather than type inference, so we do not show how to infer which lock protects a reference cell or which permission may decorate a function definition. We simply assume that the programmer can provide such information explicitly, and leave type inference as an open problem.

There is a significant body of previous work in this area, but most earlier approaches are either unsound (i.e., do not detect all race conditions) [22], deal only with finite state spaces [7, 10, 12], or do not handle mainstream shared-variable programming paradigms [1, 16]. In contrast, we aim to give a sound type system for statically verifying the absence of race conditions in a programming language with shared variables. We defer a more detailed discussion of related work to section 7.

The next section describes a first-order type system for a concurrent, imperative language. Section 3 presents the operational semantics of the language, which is the basis for the race-freeness theorem of section 4. Section 5 extends the type system with universal and existential types. Section 6 further extends the type system in order to prevent deadlocks. Section 7 discusses related work. We conclude with section 8. For the sake of brevity, we omit proofs. Moreover, we give the type systems of sections 5 and 6 without corresponding operational semantics and correctness theorems. The operational semantics are straightforward. To date, we have studied how to extend the correctness theorem of section 4 to the type systems of section 5, but only partially to that of section 6.

$$
\begin{array}{rll}
V \in \mathit{Value} & = c \mid x \mid \lambda^p x{:}t.\, e \\
c \in \mathit{Const} & = \mathit{unit} \\
x, y \in \mathit{Var} & \\
e \in \mathit{Exp} & = V \\
& \mid\ e\ e \\
& \mid\ \mathit{ref}_m e \mid\ !e \mid e := e \\
& \mid\ \mathit{fork}\ e \\
& \mid\ \mathit{new\text{-}lock}\ x{:}m\ \mathit{in}\ e \\
& \mid\ \mathit{sync}\ e\ e
\end{array}
\qquad
\begin{array}{rll}
s, t \in \mathit{Type} & = \mathit{Unit} \\
& \mid\ t \rightarrow^p t \\
& \mid\ \mathit{Ref}_m t \\
& \mid\ m \\
m, n, o \in \mathit{TypeVar} & \\
p, q \in \mathit{Permission} & = \mathcal{P}(\mathit{TypeVar})
\end{array}
$$

Fig. 1. A concurrent, imperative language.

2 First-order Types against Races

We start by considering a first-order type system focused on race conditions, and defer deadlock prevention to section 6. We formulate our type system for the concurrent, imperative language described in figure 1. The language is call-by-value, and includes values (constants, variables, and function definitions), applications, and the usual imperative operations on reference cells: allocation, dereferencing, and assignment. Although the language does not include explicit support for recursion, recursive functions can be encoded using reference cells, as described in section 2.2 below.

The language allows multithreaded programs by including the operation *fork* e which spawns a new thread for the evaluation of e. This evaluation is performed only for its effect; the result of e is never used. Locks are provided for thread synchronization. A lock has two states, locked and unlocked, and is initially unlocked. The expression $\mathit{new\text{-}lock}\ x : m\ \mathit{in}\ e$ dynamically allocates a new lock, binds x to that lock, and then evaluates e. It also introduces the type variable m which denotes the singleton lock type of the new lock. The expression $\mathit{sync}\ e_1\ e_2$ is evaluated in a manner similar to Java's `synchronized` statement [14]: the subexpression e_1 is evaluated first, and should yield a lock, which is then acquired; the subexpression e_2 is then evaluated; and finally the lock is released. The result of e_2 is returned as the result of the *sync* expression. While evaluating e_2, the current thread is said to *hold* the lock, or, equivalently, is in a *critical section* on the lock. Any other thread that attempts to acquire the lock blocks until the lock is released. Locks are not reentrant; that is, a thread cannot reacquire a lock that it already holds. A new thread does not inherit locks held by its parent thread.

2.1 The Type Rules

The type of an expression depends on a *typing environment* E, which maps program variables to types, and maps type variables to the kind *Lock* (the kind

Judgments

$E \vdash \diamond$	E is a well-formed typing environment
$E \vdash t$	t is a well-formed type in E
$E \vdash p$	p is a well-formed permission in E
$E \vdash s <: t$	s is a subtype of t in E
$E \vdash p <: q$	p is a subpermission of q in E
$E\,;p \vdash e : t$	e is a well-typed expression of type t in E with p

Rules

$$\emptyset \vdash \diamond \quad \text{(Env } \emptyset\text{)}$$

$$\frac{E \vdash t \quad x \notin dom(E)}{E, x : t \vdash \diamond} \quad \text{(Env } x\text{)}$$

$$\frac{E \vdash \diamond \quad m \notin dom(E)}{E, m :: Lock \vdash \diamond} \quad \text{(Env } m\text{)}$$

$$\frac{E \vdash \diamond}{E \vdash Unit} \quad \text{(Type Unit)}$$

$$\frac{E \vdash s \quad E \vdash t \quad E \vdash p}{E \vdash s \rightarrow^p t} \quad \text{(Type Fun)}$$

$$\frac{E \vdash t \quad E \vdash m}{E \vdash Ref_m t} \quad \text{(Type Ref)}$$

$$\frac{E, m :: Lock, E' \vdash \diamond}{E, m :: Lock, E' \vdash m} \quad \text{(Type Lock)}$$

$$\frac{\begin{array}{c} E \vdash \diamond \\ E \vdash m \quad \text{for all } m \in p \end{array}}{E \vdash p} \quad \text{(Perm)}$$

$$\frac{E \vdash p \quad E \vdash q \quad p \subseteq q}{E \vdash p <: q} \quad \text{(Subperm)}$$

$$\frac{E \vdash t}{E \vdash t <: t} \quad \text{(Sub Refl)}$$

$$\frac{\begin{array}{c} E \vdash s_1 <: t_1 \\ E \vdash t_2 <: s_2 \\ E \vdash p <: q \end{array}}{E \vdash (t_1 \rightarrow^p t_2) <: (s_1 \rightarrow^q s_2)} \quad \text{(Sub Fun)}$$

$$\frac{E \vdash \diamond}{E\,; \emptyset \vdash unit : Unit} \quad \text{(Exp Unit)}$$

$$\frac{E, x : t, E' \vdash \diamond}{E, x : t, E'\,; \emptyset \vdash x : t} \quad \text{(Exp } x\text{)}$$

$$\frac{E, x : s\,; p \vdash e : t}{E\,; \emptyset \vdash \lambda^p x{:}s.\, e : s \rightarrow^p t} \quad \text{(Exp Fun)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e_1 : s \rightarrow^p t \\ E\,; p \vdash e_2 : s \end{array}}{E\,; p \vdash e_1\ e_2 : t} \quad \text{(Exp Appl)}$$

$$\frac{E \vdash m \quad E\,; p \vdash e : t}{E\,; p \vdash ref_m e : Ref_m t} \quad \text{(Exp Ref)}$$

$$\frac{E\,; p \vdash e : Ref_m t \quad m \in p}{E\,; p \vdash\, !e : t} \quad \text{(Exp Deref)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e_1 : Ref_m t \\ E\,; p \vdash e_2 : t \quad m \in p \end{array}}{E\,; p \vdash e_1 := e_2 : Unit} \quad \text{(Exp Set)}$$

$$\frac{E\,; \emptyset \vdash e : t}{E\,; \emptyset \vdash fork\ e : Unit} \quad \text{(Exp Fork)}$$

$$\frac{\begin{array}{c} E, m :: Lock, x : m\,; p \vdash e : t \\ E \vdash p \quad E \vdash t \end{array}}{E\,; p \vdash new\text{-}lock\ x{:}m\ in\ e : t} \quad \text{(Exp Lock)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e_1 : m \\ E\,; p \cup \{m\} \vdash e_2 : t \end{array}}{E\,; p \vdash sync\ e_1\ e_2 : t} \quad \text{(Exp Sync)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e : t \\ E \vdash p <: q \quad E \vdash t <: s \end{array}}{E\,; q \vdash e : s} \quad \text{(Exp Sub)}$$

Fig. 2. The first-order type system.

of singleton lock types). The typing environment is organized as a sequence of bindings, and we use $\emptyset$ to denote the empty environment.

$$E ::= \emptyset \mid E, x : t \mid E, m :: Lock$$

We define the type system using six judgments. These judgments are described in figure 2, together with rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment $E\,; p \vdash e : t$ (read "e is a well-typed expression of type t in typing environment E with permission p"). Our intent is that, if this judgment holds, then e is race-free and yields values of type t, provided the current thread holds at least the locks described by p, and the free variables of e are given bindings consistent with the typing environment E.

The rule (Exp Fun) for functions $\lambda^p x\!:\!s.\,e$ checks that e is race-free given permission p, and then records this permission as part of the function's type: $s \rightarrow^p t$. The rule (Exp Appl) ensures that this permission is available at each call site of the function. The rule (Exp Ref) records the singleton lock type of the protecting lock as part of each reference-cell type: $Ref_m t$. The rules (Exp Deref) and (Exp Set) ensure that this lock is held (i.e., is in the current permission) whenever the reference cell is accessed. A single lock may protect several reference cells; in an obvious extension of our language, it could protect an entire record or object.

The rule (Exp Fork) typechecks the spawned expression using the empty permission, since threads never inherit locks from their parents. The rule (Exp Lock) for *new-lock* $x\!:\!m$ *in* e requires the type t of e to be well-formed in the original typing environment ($E \vdash t$). This requirement implies that t cannot contain the type variable m, and hence the *new-lock* expression cannot return the newly allocated lock. This constraint suffices to ensure that different singleton lock types of the same name are not confused. It is somewhat restrictive, but it can be circumvented with existential types, as described in section 5.2 below.

The rule (Exp Sync) for *sync* e_1 e_2 requires that e_1 yield a value of some singleton lock type m, and then typechecks e_2 with an extended permission that includes the type of the newly acquired lock. The use of this synchronization construct ensures that lock acquisition and release operations follow a stack-like discipline, which significantly simplifies the development of the type system.

The rule (Exp Sub) allows for subsumption on both types and permissions. If $E \vdash p <: q$, then any expression that is race-free with permission p is also race-free with the superset q of p.

2.2 Examples

For clarity, we present example programs using an extended language with integers, *let*-expressions, and a sequential composition operator (;). The program P_1 is a trivial example of using locks; it first allocates a lock and a reference cell protected by that lock, and then it acquires the lock and dereferences the cell. The program P_2 is slightly more complicated. It first allocates a lock and defines a function g that increments reference cells protected by that lock. It then allocates two such reference cells, and uses g to increment both of them. The type of g is $((\mathit{Ref}_m\, \mathit{Int}) \rightarrow^{\{m\}} \mathit{Int})$; it expresses that the protecting lock should be acquired before g is called.

$$
\begin{array}{lll}
P_1 & \triangleq & \textit{new-lock}\ x\!:\!m\ \textit{in} \\
 & & \quad \textit{let}\ y = \textit{ref}_m 1\ \textit{in} \\
 & & \quad\quad \textit{sync}\ x\ !y
\end{array}
\qquad
\begin{array}{lll}
P_2 & \triangleq & \textit{new-lock}\ x\!:\!m\ \textit{in} \\
 & & \quad \begin{array}{lcl} \textit{let}\ g & = & \lambda^{\{m\}} z\!:\!\mathit{Ref}_m \mathit{Int}.\ z :=!z + 1 \\ \quad y_1 & = & \textit{ref}_m 1 \\ \quad y_2 & = & \textit{ref}_m 2 \end{array} \\
 & & \quad\ \textit{in sync}\ x\ (g\ y_1; g\ y_2)
\end{array}
$$

Although the language does not include explicit support for recursion, we can encode recursion using reference cells. This idea is illustrated by the following program, which implements a server that repeatedly handles incoming requests. The core of this server is a recursive function that first allocates a new lock x_2 and associated reference cell y_2, then uses y_2 in handling an incoming request, and finally calls itself recursively to handle the next incoming request.

$$
\begin{array}{ll}
\textit{new-lock}\ x_1\!:\!m_1\ \textit{in} & \text{; Allocate a lock and a ref cell} \\
\quad \textit{let}\ y_1 = \textit{ref}_{m_1}(\lambda^{\emptyset} x\!:\mathit{Unit}.\ x)\ \textit{in} & \text{; initialized to the identity function.} \\
\quad\quad \textit{sync}\ x_1 & \text{; Acquire the lock and set the ref cell} \\
\quad\quad\quad y_1 := \lambda^{\emptyset} x\!:\mathit{Unit}. & \text{; to the recursive function.} \\
\quad\quad\quad\quad\quad\quad \textit{new-lock}\ x_2\!:\!m_2\ \textit{in} & \text{; Allocate a local lock} \\
\quad\quad\quad\quad\quad\quad\quad \textit{let}\ y_2 = \textit{ref}_{m_2} 0\ \textit{in} & \text{; and a local ref cell.} \\
\quad\quad\quad\quad\quad\quad\quad\quad \ldots & \text{; Use the local ref cell.} \\
\quad\quad\quad\quad\quad\quad\quad\quad (!y_1\ \textit{unit}); & \text{; Call the recursive function.} \\
\quad\quad\quad (!y_1\ \textit{unit}) & \text{; Start the server running.}
\end{array}
$$

The type variable m_2 denotes different singleton lock types at different stages of the execution, but these different types are not confused by the type system.

2.3 Expressiveness

Although the examples shown above are necessarily simple, the first-order type system is sufficiently expressive to verify a variety of non-trivial programs. In particular, any sequential program that is well-typed in the underlying sequential type system has an annotated variant that is well-typed in our type system. This annotated variant is obtained by enclosing the original program in the context *new-lock* $x : m$ *in sync* x [ ] (which allocates and acquires a new lock of type m), annotating each reference cell with the type m, and annotating each function definition with the permission $\{m\}$. This approach can be generalized to multithreaded programs with a coarse locking strategy based on several global locks. It also suggests how to annotate thread-local data: writing *fork* (*new-lock* $x : m$ *in sync* x e) for spawning e and using x as the lock for protecting thread-local reference cells in e.

The type system does have some significant restrictions: functions cannot abstract over lock types, and the *new-lock* construct cannot return the newly allocated lock; these restrictions are overcome in section 5. In addition, our type system is somewhat over-conservative in that it does not allow simultaneous reads of a data structure, even though simultaneous reads are not normally considered race conditions, and in fact many programs use reader-writer locks to permit such simultaneous reads [5]. We believe that adding a treatment of reader-writer locks to our type system should not be difficult.

3 Operational Semantics

We specify the operational semantics of our language using the abstract machine described in figure 3. The machine evaluates a program by stepping through a sequence of states. A state consists of three components: a lock store, a reference store, and a collection of expressions, each of which represents a thread. The expressions are written in a slight extension of the language *Exp*, called Exp_r, which includes the new construct *in-sync*. Since the result of the program is the result of its initial thread, the order of the threads in a state matters; therefore, we organize the threads as a sequence, and the first thread in the sequence is always the initial thread. We use the notation T_i to mean the ith element of a thread sequence T, where the initial thread is at index 0, and we use $T.T'$ to denote the concatenation of two sequences.

Reference cells are kept in a reference store σ, which maps reference locations to values. Locks are kept in a lock store π, which maps lock locations to either 0 or 1; $\pi(l) = 1$ when the lock l is held by some thread. Reference locations and lock locations are simply special kinds of variables that can be bound only by the respective stores. For each lock location l, we introduce a type variable o_l to denote the corresponding singleton lock type. A lock store that binds a lock location l also implicitly binds the corresponding type variable o_l with kind *Lock*; the only value of type o_l is l.

The evaluation of a program starts in an initial state with empty lock and reference stores and with a single thread. Evaluation then takes place according to the machine's transition rules. These rules specify the behavior of the various constructs in the language. The evaluation terminates once all threads have been reduced to values, in which case the value of the initial thread is returned as the result of the program. We use the notation $e[V/x]$ to denote the capture-free substitution of V for x in e, and use $\sigma[r \mapsto V]$ to denote the store that agrees with σ except at r, which is mapped to V.

The transition rules are mostly straightforward. The only unusual rules are the ones for lock creation and for *sync* expressions. To evaluate the expression *new-lock* $x : m$ *in* e, the transition rule (Trans Lock) allocates a new lock location l and replaces occurrences of x in e with l. The rule also replaces occurrences of m in e with the type variable o_l. To evaluate the expression *sync* l e, the transition rule (Trans Sync) acquires the lock l and yields the term *in-sync* l e. This term denotes that the lock l has been acquired and that the subexpression e is currently being evaluated. Since *in-sync* l [ ] is an evaluation context, subsequent transitions evaluate the subexpression e. Once this subexpression yields a value,

Evaluator

$$
\begin{array}{rcl}
eval & \subseteq & Exp \times Value \\
eval(e, V) & \Longleftrightarrow & \langle \emptyset, \emptyset, e \rangle \longmapsto^* \langle \pi, \sigma, V.unit. \cdots .unit \rangle
\end{array}
$$

State space

$$
\begin{array}{rrcl}
S \in & State & = & LockStore \times RefStore \times ThreadSeq \\
\pi \in & LockStore & = & LockLoc \rightharpoonup \{0, 1\} \\
\sigma \in & RefStore & = & RefLoc \rightharpoonup Value \\
l \in & LockLoc & \subset & Var \\
r \in & RefLoc & \subset & Var \\
T \in & ThreadSeq & = & Exp_r^* \\
f \in & Exp_r & = & V \mid f\ e \mid V\ f \mid ref_m f \mid\ !f \mid f := e \mid r := f \\
 & & \mid & fork\ e \mid new\text{-}lock\ x\!:\!m\ in\ e \\
 & & \mid & sync\ f\ e \mid in\text{-}sync\ l\ f
\end{array}
$$

Evaluation contexts

$$
\begin{array}{rl}
\mathcal{E} = & [\,] \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid ref_m \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \\
\mid & sync\ \mathcal{E}\ e \mid in\text{-}sync\ l\ \mathcal{E}
\end{array}
$$

Transition rules

$$
\begin{array}{lll}
\langle \pi, \sigma, T.\mathcal{E}[\ (\lambda^p x\!:\!t.\ e)\ V\ ].T' \rangle & \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ e[V/x]\ ].T' \rangle & \text{(Trans Appl)} \\[1ex]
\langle \pi, \sigma, T.\mathcal{E}[\ ref_m V\ ].T' \rangle & \longmapsto \langle \pi, \sigma[r \mapsto V], T.\mathcal{E}[\ r\ ].T' \rangle & \text{(Trans Ref)} \\
 & \quad \text{if } r \notin dom(\sigma) & \\[1ex]
\langle \pi, \sigma, T.\mathcal{E}[\ !r\ ].T' \rangle & \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ V\ ].T' \rangle & \text{(Trans Deref)} \\
 & \quad \text{if } \sigma(r) = V & \\[1ex]
\langle \pi, \sigma, T.\mathcal{E}[\ r := V\ ].T' \rangle & \longmapsto \langle \pi, \sigma[r \mapsto V], T.\mathcal{E}[\ unit\ ].T' \rangle & \text{(Trans Set)} \\[1ex]
\langle \pi, \sigma, T.\mathcal{E}[\ new\text{-}lock\ x\!:\!m\ in\ e\ ].T' \rangle & & \\
 & \longmapsto \langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ e[l/x, o_l/m]\ ].T' \rangle & \text{(Trans Lock)} \\
 & \quad \text{if } l \notin dom(\pi) & \\[1ex]
\langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ sync\ l\ e\ ].T' \rangle & & \\
 & \longmapsto \langle \pi[l \mapsto 1], \sigma, T.\mathcal{E}[\ in\text{-}sync\ l\ e\ ].T' \rangle & \text{(Trans Sync)} \\[1ex]
\langle \pi[l \mapsto 1], \sigma, T.\mathcal{E}[\ in\text{-}sync\ l\ V\ ].T' \rangle & & \\
 & \longmapsto \langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ V\ ].T' \rangle & \text{(Trans In-Sync)} \\[1ex]
\langle \pi, \sigma, T.\mathcal{E}[\ fork\ e\ ].T' \rangle & \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ unit\ ].T'.e \rangle & \text{(Trans Fork)}
\end{array}
$$

Fig. 3. The abstract machine.

Judgment

$\vdash S : t$ $\quad S$ is a well-typed state of type t

Rules

$$\frac{\begin{array}{c} dom(\pi) = \{l_1, \ldots, l_j\} \qquad dom(\sigma) = \{r_1, \ldots, r_k\} \\ E = o_{l_1} :: Lock,\ l_1 : o_{l_1}, \ldots, o_{l_j} :: Lock,\ l_j : o_{l_j},\ r_1 : Ref_{n_1} s_1, \ldots, r_k : Ref_{n_k} s_k \\ \forall i \in 1..k.\ E\,; \emptyset \vdash \sigma(r_i) : s_i \\ |T| > 0 \qquad \forall i < |T|.\ E\,; \emptyset \vdash T_i : t_i \end{array}}{\vdash \langle \pi, \sigma, T \rangle : t_0}$$

$$\frac{E\,; \emptyset \vdash l : m \qquad E\,; (p \cup \{m\}) \vdash f : t}{E\,; p \vdash \textit{in-sync}\ l\ f : t}$$

Fig. 4. Additional judgment and rules for typing states.

the transition rule (Trans In-Sync) releases the lock and returns that value as the result of the original *sync* expression. We say that an expression f is in a *critical section* on a lock location l if $f = \mathcal{E}[\ \textit{in-sync}\ l\ f'\ ]$ for some evaluation context $\mathcal{E}$ and expression f'.

The machine arbitrarily interleaves the execution of threads. Since different interleavings may yield different results, the evaluator *eval* is a proper relation and not simply a partial function.

We use the semantics to formalize the notion of a race condition. An expression f *accesses* a reference location r if there exists some evaluation context $\mathcal{E}$ such that $f = \mathcal{E}[\ !r\ ]$ or $f = \mathcal{E}[\ r := V\ ]$. A state has a *race condition* if its thread sequence contains two expressions that access the same reference location. A program e has a race condition if its evaluation may yield a state with a race condition, that is, if there exists a state S such that $\langle \emptyset, \emptyset, e \rangle \longmapsto^* S$ and S has a race condition.

4 Well-typed Programs Don't Have Races

The fundamental property of the type system is that well-typed programs do not have race conditions. The first component of the proof of this property is a subject reduction result stating that typing is preserved during evaluation. To prove this result, we extend typing judgments from expressions in *Exp* to expressions in Exp_r and then to machine states as shown in figure 4. The judgment $\vdash S : t$ says that S is a well-typed state yielding values of type t.

Lemma 1 (Subject Reduction). *If* $\vdash S : t$ *and* $S \longmapsto S'$, *then* $\vdash S' : t$.

Independently of the type system, locks provide mutual exclusion, in that two threads can never be in a critical section on the same lock. The judgment $\vdash_{cs} S$

Judgments

$\mathcal{M} \vdash_{cs} f$ — f has exactly one critical section for each lock in $\mathcal{M}$

$\vdash_{cs} S$ — S is well-formed with respect to critical sections

Rules

$$\frac{f = V \mid \mathit{fork}\ e \mid \mathit{new\text{-}lock}\ x : m\ \mathit{in}\ e}{\emptyset \vdash_{cs} f} \qquad \frac{\mathcal{M} \vdash_{cs} f}{\mathcal{M} \uplus \{l\} \vdash_{cs} \mathit{in\text{-}sync}\ l\ f}$$

$$\frac{\begin{array}{c}\mathcal{M} \vdash_{cs} f \\ f' = f\ e \mid V\ f \mid \mathit{ref}_m f \mid !f \\ \mid\ f := e \mid r := f \mid \mathit{sync}\ f\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \qquad \frac{\begin{array}{c}\forall i < |T|.\ \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \cdots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M}.\ \pi(l) = 1\end{array}}{\vdash_{cs} \langle \pi, \sigma, T \rangle}$$

Fig. 5. Additional judgments and rules for reasoning about critical sections.

says that at most one thread is in a critical section on each lock in S (see figure 5). According to Lemma 2, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 2 (Mutual Exclusion). *If* $\vdash_{cs} S$ *and* $S \longmapsto S'$*, then* $\vdash_{cs} S'$*.*

Lemma 3 says that a well-typed thread accesses a reference cell only when it holds the protecting lock.

Lemma 3. *Suppose that* $E\,;\emptyset \vdash f : t$*, and* f *accesses reference location* r*. Then* $E\,;\emptyset \vdash r : \mathit{Ref}_m s$ *for some lock type* m *and type* s*. Furthermore, there exists lock location* l *such that* $E\,;\emptyset \vdash l : m$ *and* f *is in a critical section on* l*.*

This lemma implies that states that are well-typed and well-formed with respect to critical sections do not have race conditions.

Lemma 4. *Suppose* $\vdash S : t$ *and* $\vdash_{cs} S$*. Then* S *does not have a race condition.*

We conclude that well-typed programs do not have race conditions.

Theorem 1. *If* $\emptyset\,;\emptyset \vdash e : t$ *then* e *does not have a race condition.*

5 Second-order Types against Races

Although the first-order type system of section 2 is applicable to a variety of multithreaded programs, there are many race-free programs that it cannot verify. This section describes extensions that allow the verification of more complex programs. These extensions rely on polymorphism and type abstraction, which are fairly easy to incorporate into a type-based approach such as ours.

Extended syntax

$$\begin{array}{rcl} V \in \mathit{Value} & = \ldots & \mid \Lambda m :: \mathit{Lock}.\, V \\ e \in \mathit{Exp} & = \ldots & \mid e[m] \\ s, t \in \mathit{Type} & = \ldots & \mid \forall m :: \mathit{Lock}.\, t \end{array}$$

Additional rules

$$\frac{E, m :: \mathit{Lock} \vdash t}{E \vdash \forall m :: \mathit{Lock}.\, t} \qquad \frac{E, m :: \mathit{Lock}\,; \emptyset \vdash V : t}{E\,; \emptyset \vdash \Lambda m :: \mathit{Lock}.\, V : (\forall m :: \mathit{Lock}.\, t)}$$

$$\frac{E, m :: \mathit{Lock} \vdash s <: t}{E \vdash (\forall m :: \mathit{Lock}.\, s) <: (\forall m :: \mathit{Lock}.\, t)} \qquad \frac{E\,; p \vdash e : (\forall m :: \mathit{Lock}.\, t) \qquad E \vdash n}{E\,; p \vdash e[n] : t[n/m]}$$

Fig. 6. Extending the first-order type system with universal types.

5.1 Polymorphism over Lock Types

The first-order type system does not permit functions parameterized by lock types. To overcome this limitation, we extend the type system to include polymorphism, as described in figure 6. The only unusual aspect of this extension is that we require the body of a polymorphic abstraction to be a value. This restriction avoids the need to annotate polymorphic abstractions with permissions, since the trivial evaluation of a value requires only the empty permission. (If needed, we can still include a non-value expression in a polymorphic abstraction by wrapping the expression in a function definition.)

Examples The following program P_3 defines a polymorphic function for incrementing a reference cell. The function abstracts over both the reference cell and the type of the lock protecting the reference cell, and the caller is responsible for acquiring that lock. The program P_4 is similar, except that the lock is acquired inside the increment function.

$$\begin{array}{rl} P_3 \triangleq & \mathit{let}\ g = \Lambda n :: \mathit{Lock}. \\ & \qquad \lambda^{\{n\}} z : \mathit{Ref}_n\, \mathit{Int}. \\ & \qquad\quad z := !z + 1 \\ & \mathit{in}\ \ \mathit{new\text{-}lock}\ x : m\ \mathit{in} \\ & \qquad \mathit{let}\ y = \mathit{ref}_m 0\ \mathit{in} \\ & \qquad\quad \mathit{sync}\ x\ (g[m]\ y) \end{array}$$

$$\begin{array}{rl} P_4 \triangleq & \mathit{let}\ g = \Lambda n :: \mathit{Lock}. \\ & \qquad \lambda^{\emptyset} w : n. \\ & \qquad\quad \lambda^{\emptyset} z : \mathit{Ref}_n\, \mathit{Int}. \\ & \qquad\qquad \mathit{sync}\ w\ (z := !z + 1) \\ & \mathit{in}\ \ \mathit{new\text{-}lock}\ x : m\ \mathit{in} \\ & \qquad \mathit{let}\ y = \mathit{ref}_m 0\ \mathit{in} \\ & \qquad\quad g[m]\ x\ y \end{array}$$

5.2 Existential Quantification over Lock Types

All our type systems require that the result type of *new-lock* $x : m$ *in* e be a well-formed type in the environment of the *new-lock* expression. This requirement forbids returning the type variable m out of the scope of its binding *new-lock* expression, and hence unfortunately excludes some useful programming patterns.

Extended syntax

$$\begin{array}{rcl} V \in \mathit{Value} &=& \ldots \mid \mathit{pack}\ m :: \mathit{Lock} = n\ \mathit{with}\ V \\ e \in \mathit{Exp} &=& \ldots \mid \mathit{open}\ e\ \mathit{as}\ m :: \mathit{Lock}, x : t\ \mathit{in}\ e \\ s, t \in \mathit{Type} &=& \ldots \mid \exists m :: \mathit{Lock}.\ t \end{array}$$

Additional rules

$$\frac{E, m :: \mathit{Lock} \vdash t}{E \vdash \exists m :: \mathit{Lock}.\ t} \qquad \frac{E, m :: \mathit{Lock} \vdash s <: t}{E \vdash (\exists m :: \mathit{Lock}.\ s) <: (\exists m :: \mathit{Lock}.\ t)}$$

$$\frac{E \vdash n \qquad E\,; \emptyset \vdash V[n/m] : t[n/m]}{E\,; \emptyset \vdash \mathit{pack}\ m :: \mathit{Lock} = n\ \mathit{with}\ V : (\exists m :: \mathit{Lock}.\ t)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e_1 : (\exists m :: \mathit{Lock}.\ t) \\ E, m :: \mathit{Lock}, x : t\,; p \vdash e_2 : s \\ E \vdash s \end{array}}{E\,; p \vdash \mathit{open}\ e_1\ \mathit{as}\ m :: \mathit{Lock}, x : t\ \mathit{in}\ e_2 : s}$$

Fig. 7. Extending the first-order type system with existential types.

For example, consider a multithreaded implementation of binary trees. To reduce lock contention, the implementation may protect each node with a separate lock. The node allocation routine thus needs to create a fresh lock, say of type m, and return a new node of type $m \times \mathit{Ref}_m \alpha \times \mathit{Ref}_m \alpha$, where α is the type of the node's children. But including m in the return type implies lifting it out of its binding *new-lock* expression, and is forbidden by the type system.

We circumvent this restriction by noting that the caller of the allocation routine does not care which singleton lock type is used to protect the node. The caller requires only that there exist some lock type m such that the node has type $m \times \mathit{Ref}_m \alpha \times \mathit{Ref}_m \alpha$. This insight suggests the use of existential types for typing such programs. It is straightforward to extend the type system with existential types, as outlined in figure 7. The type rules closely follow the conventional rules for existential types [6]. In the rule for *pack*, the lock type n is hidden and replaced with the type variable m in the resulting existential type $(\exists m :: \mathit{Lock}.\ t)$. We do not explicitly allow for renaming in the rule for *open*, since renaming can be accomplished using α-conversion, if necessary.

Examples Some of the following examples use product, sum, and recursive types, which are easily added to the type system, as in [6]. Values of these additional types are manipulated by the following operations: $\langle e_1, \ldots, e_n \rangle$ creates a value of a product type, whose components are retrieved by the operations *first*, *second*, etc.; *inLeft* creates a value of a sum type, whose component is retrieved by *asLeft*; *inRight* and *asRight* behave in a similar manner; and *fold* and *unfold* convert between a recursive type $\mu\alpha.\ t$ and its unfolding $t[(\mu\alpha.\ t)/\alpha]$.

The following expression P_5 provides a simple example of using existential types. This expression has type $\exists m :: Lock.\, (m \times Ref_m Int)$, and it returns a pair consisting of a lock and a reference cell protected by that lock. The expression P_6 opens P_5 and retrieves the value of the reference cell.

$$\begin{array}{lll}
P_5 & \triangleq & \mathit{new\text{-}lock}\ x : n\ \mathit{in} \\
& & \quad \mathit{let}\ y = \langle x, (\mathit{ref}_n 0)\rangle\ \mathit{in} \\
& & \quad\quad \mathit{pack}\ m :: Lock = n\ \mathit{with}\ y
\end{array}
\qquad
\begin{array}{lll}
P_6 & \triangleq & \mathit{open}\ P_5\ \mathit{as} \\
& & \quad m :: Lock, y : (m \times Ref_m Int)\ \mathit{in} \\
& & \quad \mathit{sync}\ \mathit{first}(y)\ !\mathit{second}(y)
\end{array}$$

For a more realistic example, we reconsider how to implement binary trees using a separate lock to protect each node. In this implementation, a leaf node is represented as *unit*, and an interior node is represented as a triple, using an existential type to hide the type of the protecting lock. The type of a binary tree is thus:

$$T \ \triangleq\ \mu\alpha.\, (Unit + \exists m :: Lock.\, (m \times Ref_m \alpha \times Ref_m \alpha))$$

Some typical routines for manipulating binary trees are:

$$\begin{array}{lll}
\mathit{leaf} & : T & \triangleq \mathit{fold}(\mathit{inLeft}(\mathit{unit})) \\
\\
\mathit{alloc\text{-}node} & : T \rightarrow^{\emptyset} (T \rightarrow^{\emptyset} T) & \triangleq \lambda^{\emptyset} l : T. \\
& & \quad \lambda^{\emptyset} r : T. \\
& & \quad\quad \mathit{new\text{-}lock}\ x : n\ \mathit{in} \\
& & \quad\quad\quad \mathit{pack}\ m :: Lock = n\ \mathit{with} \\
& & \quad\quad\quad\quad \mathit{fold}(\mathit{inRight}(\langle x, \mathit{ref}_n l, \mathit{ref}_n r\rangle)) \\
\\
\mathit{left\text{-}child} & : T \rightarrow^{\emptyset} T & \triangleq \lambda^{\emptyset} x : T. \\
& & \quad \mathit{open}\ \mathit{asRight}(\mathit{unfold}(x)) \\
& & \quad\quad \mathit{as}\ m :: Lock, y : (m \times Ref_m T \times Ref_m T) \\
& & \quad\quad \mathit{in}\ \mathit{sync}\ \mathit{first}(y)\ !\mathit{second}(y)
\end{array}$$

6 Preventing Deadlocks

The type systems described so far ensure that well-typed programs do not have race conditions. However, these programs may still suffer from other errors, in particular deadlocks.

We formalize the notion of deadlock using the operational semantics of figure 3. An expression f *requests* a lock l if $f = \mathcal{E}[\ \mathit{sync}\ l\ e\ ]$. A state is *deadlocked* if there is a cycle of threads in the state such that each thread requests a lock held by the next thread in the cycle. More precisely, a state $S = \langle \pi, \sigma, T\rangle$ is deadlocked if there exist lock locations $l_0, \ldots, l_n$ and indices $d_0, \ldots, d_{n-1}$ of T such that $n > 0$, $l_0 = l_n$, and for each $0 \leq i < n$, thread T_{d_i} is in a critical section on l_i and requests lock l_{i+1}. A program e may deadlock if its evaluation may yield a deadlocked state, that is, if there exists a deadlocked state S such that $\langle \emptyset, \emptyset, e\rangle \longmapsto^* S$.

Judgments (in addition to those of figure 2)

$E \vdash m :: (\overline{m}_1, \overline{m}_2)$	m is in the interval $(\overline{m}_1, \overline{m}_2)$ in E
$E \vdash m_1 \prec m_2$	m_1 is less than m_2 in E
$E \vdash (\overline{m}_1, \overline{m}_2)$	$(\overline{m}_1, \overline{m}_2)$ is a well-formed, non-empty interval in E

Rules (partial list)

$$\frac{E \vdash \diamond \qquad m \notin dom(E) \qquad E \vdash (\overline{m}_1, \overline{m}_2)}{E, m :: (\overline{m}_1, \overline{m}_2) \vdash \diamond}$$

$$\frac{E \vdash \diamond \qquad \forall m_1 \in \overline{m}_1.\ \forall m_2 \in \overline{m}_2.\ E \vdash m_1 \prec m_2}{E \vdash (\overline{m}_1, \overline{m}_2)}$$

$$\frac{E, m :: (\overline{m}_1, \overline{m}_2), E' \vdash \diamond}{E, m :: (\overline{m}_1, \overline{m}_2), E' \vdash m :: (\overline{m}_1, \overline{m}_2)}$$

$$\frac{\overline{m}_1 \subseteq \overline{m}_2 \qquad E \vdash L_2 \prec L_1 \text{ or } L_1 = L_2}{E \vdash \langle \overline{m}_1, L_1 \rangle <: \langle \overline{m}_2, L_2 \rangle}$$

$$\frac{E \vdash \diamond \quad E \vdash m :: (\overline{m}_1, \overline{m}_2) \quad m_1 \in \overline{m}_1}{E \vdash m_1 \prec m}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e : \mathit{Ref}_m t \qquad m \in \overline{m}}{E\,; \langle \overline{m}, L \rangle \vdash !e : t}$$

$$\frac{E \vdash \diamond \quad E \vdash m :: (\overline{m}_1, \overline{m}_2) \quad m_2 \in \overline{m}_2}{E \vdash m \prec m_2}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e_1 : \mathit{Ref}_m t \qquad E\,; \langle \overline{m}, L \rangle \vdash e_2 : t \qquad m \in \overline{m}}{E\,; \langle \overline{m}, L \rangle \vdash e_1 := e_2 : \mathit{Unit}}$$

$$\frac{E \vdash \diamond \quad E \vdash m}{E \vdash m \prec \top}$$

$$\frac{E\,; \langle \emptyset, \bot \rangle \vdash e : t}{E\,; \langle \emptyset, \top \rangle \vdash \mathit{fork}\ e : \mathit{Unit}}$$

$$\frac{E \vdash \diamond \quad E \vdash m}{E \vdash \bot \prec m}$$

$$\frac{E, m :: (\overline{m}_1, \overline{m}_2), x : m\,; p \vdash e : t \qquad E \vdash p \quad E \vdash t}{E\,; p \vdash \mathit{new\text{-}lock}\ x : m :: (\overline{m}_1, \overline{m}_2)\ \mathit{in}\ e : t}$$

$$\frac{E \vdash m_1 \prec m \qquad E \vdash m \prec m_2}{E \vdash m_1 \prec m_2}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e_1 : m \qquad E \vdash L \prec m \qquad E\,; \langle \overline{m} \cup \{m\}, m \rangle \vdash e_2 : t}{E\,; \langle \overline{m}, L \rangle \vdash \mathit{sync}\ e_1\ e_2 : t}$$

Fig. 8. Extending the type system for deadlock elimination (highlights).

In practice, deadlocks are commonly avoided by imposing a strict partial order on locks, and respecting this order when acquiring locks [5]. We capture this discipline by embodying it in an extension of our type system.

Our extended type system relies on annotations that specify a lock ordering. Whenever we introduce a singleton lock type, we must specify an appropriate order between that lock type and the other lock types in the program. If $\overline{m}_1$ and $\overline{m}_2$ are sets of lock types, then we use the notation $m :: (\overline{m}_1, \overline{m}_2)$ to mean that the lock type m is greater than each lock in $\overline{m}_1$ and is less than each lock in $\overline{m}_2$. Thus the interval $(\overline{m}_1, \overline{m}_2)$ specifies a kind. In the extended language, we use kinds of the form $(\overline{m}_1, \overline{m}_2)$ instead of the kind *Lock*:

$$
\begin{array}{rcll}
V \in \mathit{Value} & = \dots & | & \Lambda m :: (\overline{m}_1, \overline{m}_2).\, V \\
 & & | & \mathit{pack}\; m :: (\overline{m}_1, \overline{m}_2) = n \;\mathit{with}\; V \\
e \in \mathit{Exp} & = \dots & | & \mathit{new\text{-}lock}\; x : m :: (\overline{m}_1, \overline{m}_2) \;\mathit{in}\; e \\
 & & | & \mathit{open}\; e \;\mathit{as}\; m :: (\overline{m}_1, \overline{m}_2), x : t \;\mathit{in}\; e \\
s, t \in \mathit{Type} & = \dots & | & \exists m :: (\overline{m}_1, \overline{m}_2).\, t \\
 & & | & \forall m :: (\overline{m}_1, \overline{m}_2).\, t
\end{array}
$$

The type system ensures that locks are acquired in the appropriate order using the notion of a *locking level*. A locking level L is either a particular lock type, in which case any greater lock can be acquired, or $\bot$, in which case all locks can be acquired, or $\top$, in which case no lock can be acquired. We extend permissions to include a locking-level component, so a permission is a pair of a lock set and a locking level. The trivial, empty permission is $\langle \emptyset, \top \rangle$, and the initial permission (of forked threads and of the main program) is $\langle \emptyset, \bot \rangle$.

We extend the typing environment E to map type variables to intervals $(\overline{m}_1, \overline{m}_2)$. The judgment $E \vdash m_1 \prec m_2$ expresses that m_1 is less than m_2; the judgment $E \vdash (\overline{m}_1, \overline{m}_2)$ expresses that $(\overline{m}_1, \overline{m}_2)$ is a well-formed, non-empty interval; and the judgment $E \vdash m :: (\overline{m}_1, \overline{m}_2)$ expresses that the lock type m is in the interval $(\overline{m}_1, \overline{m}_2)$. In a well-formed environment, the ordering constraints on lock variables induce a strict partial order.

The necessary modifications to the type rules are outlined in figure 8. Most of the rules are straightforward adaptations of earlier rules. The rule for *fork* initializes a newly spawned thread with the locking level $\bot$, since that thread is free to acquire any lock. The rule for *sync* ensures that locks are acquired in increasing order. Collectively, the type rules check that threads respect the strict partial order on locks, as required by the discipline for preventing deadlocks.

7 Related Work

Race conditions and deadlocks have been studied for decades, for example in the program-verification literature. In this section, we mention some of the work most closely related to ours.

Warlock [22] is a system for detecting race conditions and deadlocks statically. Its goals are similar to those of our type system. The major differences are that Warlock is an implemented system applicable to substantial programs, and that

Warlock may fail to detect certain race conditions. This unsoundness is partly due to the target language (ANSI C), and partly due to two other difficulties that are overcome by our system. First, Warlock works by tracing execution paths, but it fails to trace paths through loops or recursive function calls. Second, Warlock appears to merge different locks of the same type, and so may fail to detect inconsistent locking.

Aiken and Gay [2] also investigate static race detection, in the somewhat different setting of SPMD programs. They present a system that has been used successfully on a variety of SPMD programs. Synchronization in these programs is performed using barriers. Since a barrier is a global operation not associated with any particular variable in the program, they do not develop machinery for tracking the association between reference cells and their protecting locks.

A number of analyses have been developed for concurrent languages such as CML [20]. Nielson and Nielson [19] present an analysis that predicts process and channel utilization and uses this information for optimization. Their analysis is based on the notion of behaviors, which are similar to our permissions. Colby [9] also presents an analysis that infers information about channel usage. Neither work treats race conditions or deadlocks.

Kobayashi [16] presents a first-order type system for a process calculus. His type system has a deadlock-free subset, and uses the notion of time tags, which are similar to our locking levels. Although Kobayashi considers some sophisticated determinism properties, he does not address race conditions directly.

Abramsky, Gay, and Nagarajan [1] present another type-based technique for avoiding deadlocks. Their work is based on interaction categories inspired by linear logic. It emphasizes issues of type structure, rather than their application to a specific programming language.

Dwyer and Clarke [11] describe a data-flow analysis for verifying certain correctness properties of concurrent programs, for example mutual exclusion on particular resources. The authors suggest that their analysis is not well suited for detecting global properties such as deadlock. Avrunin *et al.* [4] describe a toolset for analyzing concurrent programs. This toolset has been used for detecting race conditions and deadlocks in a variety of benchmarks, on a case-by-case basis.

Savage *et al.* [21] describe Eraser, a tool for detecting race conditions and deadlocks dynamically (rather than statically, as in our method). Although quite effective, Eraser may fail to detect certain race conditions and deadlocks because of insufficient test coverage. In general, static checking and testing are complementary, and they should both be used in the development of reliable software. Hybrid approaches (like that of the Cilk Determinator [8]) seem promising.

There is a large amount of work on model-checking of concurrent programs, particularly focused on finite-state systems (*e.g.*, [7, 10, 12]). Recently, Godefroid has applied model-checking techniques to C programs [13]; his approach, stateless state-space exploration, relies on dynamic observation rather than static analysis.

The permissions that we use are similar to effects [15, 17, 18] in that the permission of an expression constrains the effects that it may produce. Much work has been done on effect reconstruction [3, 23–25]. It may be possible to adapt these inference methods in our setting in order to remove the need for explicit lock annotations.

8 Conclusions

This paper describes how a type system can be used for avoiding two major pitfalls of multithreaded programming, namely race conditions and deadlocks. Our approach requires annotating programs with locking information. We believe that this information is usually known to competent programmers and often implicit in documentation. However, it may be worthwhile to investigate algorithms for inferring the annotations. Such algorithms would be helpful in tackling larger examples and in extending our techniques to programming languages more realistic than the one treated in this paper. Also helpful would be a mechanism for escaping from our type system when it proves too restrictive. We leave those issues for further work.

For sequential languages, standard type systems provide a means for expressing and checking fundamental correctness properties. We hope that type systems such as ours will play a similar role in the realm of multithreaded programming.

Acknowledgments

Comments from Mike Burrows, Rustan Leino, and Mark Lillibridge were helpful for this work.

References

1. S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, 1997.
2. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.
3. T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.
4. G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, 1991.
5. A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
6. L. Cardelli. Type systems. *Handbook of Computer Science and Engineering*, pages 2208–2236, 1997.
7. A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.

8. G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.
9. C. Colby. Analyzing the communication topology of concurrent programs. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, 1995.
10. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.
11. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.
12. L. Fajstrup, E. Goubault, and M. Raussen. Detecting deadlocks in concurrent systems. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
13. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.
14. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
15. P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 303–310, 1991.
16. N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 128–139, 1997.
17. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.
18. F. Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2):344–345, 1996. Invited position statement for the Symposium on Models of Programming Languages and Computation.
19. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 84–97, 1994.
20. J. H. Reppy. CML: a higher-order concurrent language. In *ACM '91 Conference on Programming Language Design and Implementation.*, pages 293–305, 1991.
21. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
22. N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.
23. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
24. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 188–201, 1994.
25. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.